A real-time operating system

Roker, I. G. R. J.

Award date:
1986

Awarding institution:
University of Bath

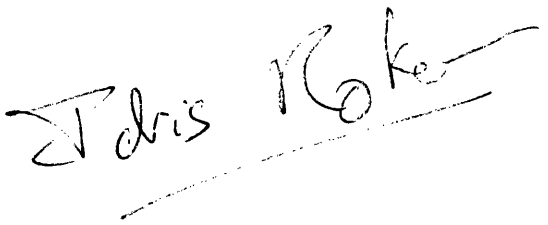
A REAL-TIME OPERATING SYSTEM

submitted by I. G. R. J. Roker

for the degree of Ph. D.

of the University of Bath

1986



Bath, September 1986

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

To **Lil**

and **Marion**

Contents

Summary

The fields of real-time modelling and real-time digital control methods have grown rapidly in recent years. This growth has not been matched by the development of suitable environments for the designing, coding, testing and running of real-time tasks.

This dissertation examines the advantages which would be gained from being able to develop and run real-time tasks in the environment provided by a multi-user, multi-tasking operating system and develops a specification for a general-purpose operating system to run on a multi-processor computer system which would support the execution of real-time tasks. The implementation of this specification based on the UNIX operating system is described.

An appraisal of the possible applications for such a system is given, along with an indication of future enhancements which could be applied to the system and references to recent papers in the field of real-time programming applications.

Abbreviations and Typographical Conventions

This section lists the principal abbreviations which are used in this thesis. Others are given on their first occurrence in the text. In accordance with standard English usage, masculine pronouns are used to denote objects which can be either masculine or feminine.

Function names of the form *fname()* represent routines written in the C programming language, while function names of the form *fname:* represent routines written in assembly language.

UNIX® is a registered trademark of AT&T in the USA and other countries.

CPU	Central Processing Unit
DIL	Dual-in-line
DMA	Direct Memory Access
EPROM	Erasable Programmable Read-Only Memory
FIFO	First In - First Out Register
I/O	Input/Output
kb	kilobyte
RAM	Random Access Memory
rpm	revolutions per minute
Mb	megabyte
MMU	Memory Management Unit
ms	millisecond
PAL	Programmable Array Logic chip
TTL	Transistor-Transistor Logic

Chapter 1

Introduction

1.1 The Development of Microprocessor-based Systems

In 1971 Intel Corporation introduced the 4-bit *4004* microprocessor[48]. This device was originally intended for use in calculators only, but, due to the liquidation of the firm which commissioned the design, it was marketed as a general-purpose processor chip.

Since then, the growth of the microprocessor industry has been phenomenal. Eight-bit microprocessors such as the M6800[27] from Motorola and the Z80[47] from Intel soon appeared and were in common use by the end of the 1970's. In the 1980's 16-bit and 32-bit microprocessors with sophisticated architectures have been developed and marketed which exhibit processing powers far in excess of the large mainframe computers of the 1960's. Examples of these are the M68010[29] and M68020[30] from Motorola.

Along with the increase in the availability and power of microprocessors, there has also been a corresponding increase in the availability and power of support chips for microprocessor-based systems. High-speed, large-capacity memory chips and intelligent peripheral controllers are now readily available, as are microprocessor-based intelligent peripherals.

Due to the low cost of most of these components, there has been a rapid increase in the number of microprocessor-based computer systems on the market. With the proliferation of minicomputers, intelligent workstations and home computers a considerable number of people are now

"computer literate" and a larger number have had some contact with computing systems.

The growth of the microprocessor industry has also changed the face of many areas of engineering. Many products are appearing which are microprocessor-based and these generally offer increased flexibility and reliability over their older hardwired counterparts.

Such growth in hardware production could not have been sustained without a corresponding growth of software for microprocessor systems. In 1974 CP/M[46], the first operating system aimed specifically at microprocessor systems, was written by Intel Corporation for the 8080 and Z80 microprocessors. Since then, a number of operating systems, programming environments and high-level languages have been made available for microprocessors.

1.2 The Development of Control Systems

The related areas of control, measurement and simulation have developed steadily since the First World War. Although research in these areas has been driven to a large extent by the interest in defence systems and spaceflight, many of the results of this research have been incorporated into less esoteric applications.

The methodology of control system design has followed the availablity of electronic components, benefitting from the development of the transistor, the integrated circuit and, lately, the microprocessor. The incorporation of programmable elements into control systems has facilitated the design and production of adaptive and real-time controllers which exhibit hitherto unobtainable levels of performance.

Although the theory of real-time digital control systems has kept pace with the availability of sophisticated hardware, there have been

relatively few attempts to 'modernise' the design environment for such systems. Much of today's control software is written in assembly languages for reasons of speed and the low availability of high-level languages with real-time features.

Many control applications would benefit from some measure of concurrency due to the number of operations which must be performed in each sample period (e.g. in excess of 10^{10} operations for aircraft simulation[33]) and also because the operations involved are often 'logically concurrent' and are therefore easier to simulate by concurrent programming. To programme concurrent operations in an assembly language for a standalone embedded microprocessor controlled system is, however, an extremely difficult and time-consuming process and is therefore costly.

Other features which would be potentially useful in real-time control systems include the ability to easily monitor the control task, the ability to easily transfer the control software to the target system, the ability to employ high-level debugging tools to the control task in both the design and the running stages and the ability to transfer information from a database to the control task in an efficient manner. All these features involve lengthy and difficult programming, however.

1.3 Attempts to Develop Better Environments for Real-Time Programming

Most attempts to overcome the limitations of real-time programming environments discussed in the previous section have involved the development of high-level languages with features which would be useful in real-time applications (e.g. concurrency, strict type checking, etc.). Some of these languages are completely new (e.g. **Ada**[35] and PROGRESS[48]) while others are attempts to modify and extend existing

high-level languages to cope with the requirements of real-time software (e.g. ConCurrent C[31], Concurrent Pascal[34] and BCS[93]) by adding new data types and functions.

The development of suitable high-level languages, while going a long way to alleviate the difficulties involved in developing real-time software, does not, however, fully address all of the associated problems. For example, although a language which has concepts of concurrent operations will greatly simplify the writing of concurrent software, if the resulting programme is run on a computer system whose underlying software and/or hardware does not support concurrency, there will be no increase in the efficiency of the software. Indeed, in many cases the software will run less efficiently, as the computing system will require some of the available resources in order to simulate the concurrent operations.

A number of designs have come forward for software and hardware which will efficiently support high-level real-time languages. These include MERT[65], PEST[39], REBUS[1] and No. 5 ESS[16], amongst others, some of which have been given no names[4,18,19,21,34,40]. A large number of these systems have been designed for specific applications (e.g. No. 5 ESS for digital telephone switching systems) while others have been specifically designed for particular computer systems (e.g the M2140 from English Electric[19]). Other systems, while being more portable, have contained restrictions regarding languages[9], task priority[39], etc. Because of these restrictions, such systems have not proved to be suitable as a basis for the development of portable, general-purpose real-time software, and many proposed systems have never progressed beyond the design stage.

1.4 A Proposal for a Real-Time System

It has been shown above that there exists a need for a system which will support the efficient development and execution of real-time tasks and which will also be portable across a large range of computer systems and applications.

It is proposed here to develop a portable multi-user, multi-tasking operating system which will provide the capability of running both high-priority, real-time tasks and low-priority, normal tasks in parallel. This will provide the opportunity of having low-level monitor tasks to support the real-time tasks as well as allowing both the writing and the running of real-time software on the same system.

The system will be based on a well-known, popular and readily available operating system, in order to be able to capitalize on software which will already have been written and tested for use on such a system.

Although the system will support real-time high-level languages, the lack of such languages on a particular system should not preclude the use of the real-time facilities which the system offers. The system facilities should, therefore, be easily accessible by other means. This applies equally to the normal operating system facilities as well (e.g. I/O) and would therefore provide a real-time task with easy access to high-level operations.

The considerations surrounding the design of such an operating system based on the commercially available UNIX[TM] operating system is the subject of the remainder of this thesis.

Chapter 2

The Hardware Elements

This chapter gives a general description of the main hardware elements which make up the computing system used for this project. The system is based around the M68000 family of microprocessors, in particular the M68000 and the M68010. Some of the devices mentioned were only used on the original multi-board version of the system, but these are included where they provide some insight into the development of the single-board system on which most of the work for the project was done.

2.1 The M68000 Microprocessor

2.1.1 General Features

The M68000[2a] is a 16 bit microprocessor which was introduced by Motorola Inc. in 1977. It has 8 internal general-purpose data registers (D0-D7) and 8 internal general-purpose address registers (A0-A7), each of 32 bits. Its external data bus is only 16 bits wide, however, so full 32 bit data fetches have to occupy two fetch cycles. Its address bus is 24 bits wide and uses only the lower 24 bits of the address registers and program counter. The direct addressing range of the M68000 is therefore over 16Mb.

In addition to the general purpose registers, the M68000 contains a 32 bit programme counter and a 16 bit status register. The status register is divided into two 8 bit sections. The upper byte is called the system byte and indicates the state of the processor. The lower byte is called the user byte (or the condition codes register) and reflects the

result of the last instruction executed. The address register A7 is also used as a stack pointer.

The M68000 contains instructions to operate on a range of data types. Its normal word length is 16 bits, and most operations reflect this. It can, however, also perform operations using bytes (8 bits) and long words (32 bits) by, in the first case, using only half of its data bus or, in the second case, performing two data read or write cycles. Manipulation of individual bytes is possible and the ADD and SUBTRACT instructions can access bytes as two binary-coded decimal words. All instructions are one word long but some addressing modes may require between one and four extension words to completely describe the operation.

Data can be addressed in a variety of ways using the M68000. Data in the address or data registers can be addressed directly and this is a particularly efficient form of addressing as the address is encoded in the instruction and no external read or write cycles are necessary to access the data. Data in external memory locations can be accessed by giving its absolute address (either as an extension to the instruction or in an address register) or by providing an offset relative to the programme counter. If the address is provided in an address register, it is possible to provide a fixed offset as an extension to the instruction or to use another register as an index register in order to modify the address. In addition to this, the instruction can cause the address register to automatically decrement before the address is used in the instruction, or to automatically increment after the address is used in order to provide easy access to blocks of data.

All I/O devices in an M68000 system are memory mapped so no separate I/O bus is necessary. Interrupts are prioritised into seven

levels, of which the highest level is non-maskable. The M68000 runs in either of two privilege states and interrupt processing always takes place in the higher ("supervisor") privilege state. There is a separate stack pointer for each privilege state (although each is accessed through A7) and this ensures that ordinary tasks running in the "user" state are not affected by hardware interrupts and other exceptions.

2.1.2 Data Organisation

Data is stored in the general-purpose registers of the M68000 as long words of 32 bits. This data can also be accessed, howver, as 16 bit words or 8 bit bytes (address registers do not support byte operations). When less than the full long word is accessed, the low order bits are always used. When the low order bits of a data register are accessed, the upper bits remain unchanged. In an address register, all operands are sign-extended to 32 bits before being used.

In memory, data can be stored and accessed as bit patterns, bytes, words or long words but words and long words must be accessed on even addresses. In word data, the upper byte lies on an even address *n* and the lower byte lies on an odd address *n+1*. Long word data is organised as two adjacent words with the higher order word at an (even) address *n* and the lower order word at (even) address *n+2*. This organisation is illustrated in Fig. 2.1.

Instructions for the M68000 are arranged in memory as words in ascending order. Each instruction in the instruction set occupies one word, but, depending on the addressing mode used for the source or destination operand, a number of extension words may have to follow the instruction word. The format of these extension words is described below for each addressing mode.

2.1.3 Addressing Modes

The M68000 supports 14 different addressing modes which fall into 6 categories.

2.1.3.1 Implied Addressing

Some instructions inherently refer to certain registers. For example, an instruction is available which allows a value to be placed in the user mode stack pointer. No extension words are required for this addressing mode.

2.1.3.2 Immediate Data Addressing

This addressing mode does not supply an address at all, as the operand is contained in the one or two extension word(s) of the instruction

2.1.3.3 Direct Addressing

Using this mode, the operand is in the specified data or address register. No extension words are required.

2.1.3.4 Absolute Addressing

In this mode the address of the operand is given in the following one or two extension word(s). If the operand is in the low area of memory, it may be possible to specify its address by using only 16 bits so only one word of extension is necessary. Frequently used addresses are therefore normally placed in low memory. The entire memory map may be accessed if two extension words are used, however.

2.1.3.5 Indirect Addressing

This addressing mode uses an address register to provide the base address of the operand. If the address register contains the address of the operand itself, no extension words are required for the instruction. In this case, it is also possible to have the address register decremented before its value is used (predecrement indirect addressing) or incremented after its value is used (postincrement indirect addressing). The register is always incremented/decremented by the size of the operand it references so care must be taken when storing byte data in the same block as multi-byte data as an access to a single byte will cause the register to become misaligned for multi-byte accesses (which must always occur on an even boundary). Byte data is stored and retrieved from the progamme stack (referenced by A7) on even boundaries, so the condition is relaxed in this special case.

It is possible to provide a fixed offset to the value in the address register. The offset (displacement) is contained in a single extension word and is sign-extended to 32 bits and added to the value in the address register to provide the address of the operand. It is also possible to provide a variable offset (index) to the value in the address register. This also requires one extension word of which the lower byte provides an 8 bit displacement which is sign-extended to 32 bits before use. The upper byte specifies an address or data register which is to be used as the index register and the length of the value contained therein which can be the full 32 bits or only the lower 16 bits, in which case the value is sign-extended before use. The address of the operand is calculated from the sum of the address register, index register and displacement.

2.1.3.6 Relative Addressing

This mode uses the programme counter as an address register to which a displacement or index may be added in a single extension word as described above. Using this mode, programmes can be written which are independent of the their absolute location in memory.

2.1.4 The M68000 Instruction set

The instruction set of the M68000 contains some 54 basic instruction types which fall into seven categories. These categories are described below.

2.1.4.1 Data Movement Instructions

The data movement instructions provide the basic method for the acquisition and manipulation of data. It is possible to transfer data from memory to register, register to memory, memory to memory and register to register. In addition, more complex operations are provided which allow the exchange of registers, the linking of successive areas of a stack frame, the movement of data in a register to *alternate* bytes of memory (to facilitate movement of data to peripheral devices) and movement of addresses. Most operations can be used for operands of all sizes.

2.1.4.2 Arithmetic Instructions

The M68000 can perform a range of integer arithmetic functions on byte, word or long word data. The basic operations (addition, subtraction, multiplication and division) are provided as well as operations for comparing two operands, clearing an operand and two's complement negation of an operand. The multiplication and division

instructions take signed or unsigned operands and produce a 32 bit product from two 16 bit multiplicands or a 16 bit quotient and 16 bit remainder from a 32 bit dividend and 16 bit divisor.

Multiprecision arithmetic is possible using extended versions of the add, subtract and negate instructions which use the "carry" value (if any) from the previous operation. Add, subtract and negate instructions also exist for binary-coded decimal data.

Two specialised instuctions fall into this category. The first is the "test" instruction (TST) which compares its operand with zero and the second is the "test and set" instruction (TAS) which performs the same operation but sets the highest order bit in its (byte) operand.

This latter operation is performed using an indivisible "read-modify-write" cycle and is therefore very useful in the synchronisation of processors in a multi-processor system.

2.1.4.3 Logical Instructions

The logical instructions provide bitwise logical **AND**, **OR**, **EXCLUSIVE-OR**, and **NOT** operations on all sizes of data.

2.1.4.4 Shift and Rotate Instructions

Operands in registers or in memory can be shifted or rotated either left or right by a fixed or variable number of bit positions. (Memory resident operands are 16 bits only and only a single bit shift or rotate is allowed). For variable shifts, the count is specified in a data register.

2.1.4.5 Bit Manipulation Instructions

Operations are available to test, clear, set and change individual bits of an operand in memory or in a data register.

2.1.4.6 Programme Control Instructions

The value in the programme counter can be altered in an absolute or relative manner by **JUMP** and **BRANCH** instructions respectively which transfer control to another area of the programme. In addition, a return address is stored by instructions which jump or branch to a subroutine and is retrieved by the corresponding **RETURN** instruction at the end of the subroutine.

2.1.4.7 System Control Instructions

System instructions are executed in supervisor mode and provide facilities for resetting external devices, stopping programme execution until an interrupt occurs, returning to the main routine after exception processing, and changing the state of the processor.

2.1.5 Hardware Signals

The M68000 comes in a 64 pin DIL package and provides external signals which fall into the categories described below.

2.1.5.1 General Inputs

General Inputs to the M68000 are the power input and the clock. The clock input is TTL compatible and is internally buffered. Versions of the M68000 use 8MHz, 10MHz and 12MHz clock frequencies.

2.1.5.2 Data Bus

This is a 16 bit bidirectional three-state bus. It provides a path for general-purpose data in normal operation and carries the interrupt vector during an interrupt acknowledge cycle.

2.1.5.3 Address Bus

This is a 23 bit (A1-A23) bidirectional three-state bus which provides the bits 1-23 of the address for the current operation in all bus cycles except the interrupt cycle. Bit 0 of the address is not present on the address bus since upper and lower byte operations are defined by the upper and lower data strobes described below. During an interrupt acknowledge cycle, the low order 3 lines (A1-A3) provide the interrupt priority level (0-7) and the remaining lines are high.

2.1.5.4 Asynchronous Bus Control

Five signals fall into this category. The **read-write** line defines the current data bus transfer as being a read or a write cycle. The **address strobe** indicates when the state of the address lines indicate a valid address. The **upper and lower data strobes** indicate which portion of the data bus carries valid data. When the upper data strobe is low, bits 8-15 of the data bus are valid, whereas the lower data strobe indicates that bits 0-7 of the data bus are valid. The two signals act independently of each other so full 16 bit operations are possible. All of the above signals are outputs.

One input signal also assists the bus control. When a low signal appears on the **data acknowledge** line, it indicates to the processor that the transfer is complete. In a write cycle, this signals indicates to the processor that the memory has latched the data so the processor

terminates the cycle. In a read cycle it indicates that the data on the bus is valid so the processor latches it and then terminates the cycle.

2.1.5.5 Bus Arbitration Control

Under normal operation, the M68000 controls all bus cycles. Other devices on the bus can also take control of the bus, however, and the processor halts when another device is bus master. The **bus request** line is an input to the processor and, when it is low, indicates that some other device wants control of the bus. When the processor receives this signal, it outputs a low signal on the **bus grant** line as soon as possible. This indicates to the external device that the bus will be available after the completion of the current bus cycle. When the requesting device receives this signal, it puts a low signal on the **bus grant acknowledge** input of the processor. This line remains low as long as the external device controls the bus.

As there is only one set of arbitration control signals, external arbitration circuitry is necessary on systems where more than one external device can become bus master.

2.1.5.6 Interrupt Control

Three input pins encode the priority level of the device which is currently requesting an interrupt. If the interrupt is granted, this level is put on pins A1-A3 as an "address" prior to reading the interrupt vector.

2.1.5.7 Processor Status

Three output pins indicate the processor's current privilege state and whether the current bus cycle is in the programmme or data area of memory. A special code indicates an interrupt acknowledge cycle.

2.1.5.8 System Control

The **reset** line is a bidirectional line which, as an input, resets the processor when it is low and, as an output, allows the processor to reset all external devices connected to it. The **halt** line is also bidirectional. As an input, it causes the processor to halt at the end of the current bus cycle and make all its control signals inactive. As an output, it indicates that the processor has encountered a serious error condition from which it could not recover and has therefore halted. The **bus error** input line informs the processor that it should abort the current bus cycle as an error has occurred. If the processor receives a bus error signal while processing another bus error, it halts.

2.1.5.9 M6800 Peripheral Control

This final group of signals allows the M68000 processor to interface to peripheral devices which were designed for the M6800 series microprocessors. These were earlier 8 bit processors with a synchronous bus and the three signals provided by the M68000 are the equivalent of the bus control signals on the earlier processors.

2.1.6 Bus Operation

Data transfer in an M68000 system uses the address bus, data bus and control signals described above. Some of the control signals are active when there is a high voltage on the control line, while others are

active when the voltage on the line is low. To avoid confusion in the description of their operation, signals will be defined as being *asserted* when they are in their active state irrespective of whether that state is defined by a low or a high voltage. Similarly, when a signal is inactive, it will be described as being *negated*.

As the bus operation is asynchronous, it is impossible to say what state a particular signal will be in at a particular point in the middle of a bus cycle. For this reason, the device in the system which is bus master for a particular cycle is responsible for ensuring that all signals are in a known state both at the start and end of a bus cycle.

2.1.6.1 Read Cycles

In read cycles, the processor acquires data from external memory or a memory-mapped peripheral device. The processor always reads bytes of data and asserts the upper or lower data strobe to indicate which byte of a multi-byte transfer it is attempting to access.

At the beginning of a read cycle, the address bus and function code (i.e. processor status) outputs are in the high impedance state, the read-write line indicates a read cycle and the address strobe and data strobes are negated. The processor then places the required address on the address bus and outputs the required function code. When the address bus is in a stable state, the address strobe is asserted and the upper and lower data strobes are asserted as required.

On receipt of a valid address, the selected device places the requested data on the data bus and asserts the data acknowledge signal. The device must then wait until the address strobe or the data strobe is negated before removing the data from the data bus.

2.1.6.2 Write Cycles

Write cycles begin in the same way as read cycles. When the valid address and function codes have stabilised, the address strobe is asserted and the read-write line is set to indicate a write operation. After this, the data is put onto the data bus and the data strobe(s) are asserted when this bus is stable. The data remains on the bus until the bus master receives the data acknowledge signal when the address and data strobes are negated.

2.1.6.3 Read-Modify-Write Cycles

These bus cycles are only used by the test and set instruction (TAS). This cycle is very much like a read cycle followed immediately by a write cycle, except that, at the end of the read half of the cycle, only the data strobes are negated. As the address strobe remains asserted, the read and write halves of the cycle are indivisible.

2.1.6.4 Bus Error Operation

The bus architecture described above requires that the external device provide a signal to the bus master at some stage in the cycle. It may happen that such a signal never occurs due to faults in the external device or in the address provided and external circuitry is therefore necessary to signal the processor to abort the current bus cycle by asserting the bus error signal.

If the bus error signal is asserted while the halt line is inactive, the processor initiates an exception sequence described below. If the halt pin is active, the processor completes the bus cycle and ceases to execute instructions, leaving all its output signals in the high impedance state. When the halt signal is removed, the processor will re-run the

previous bus cycle using the same address, function codes and data (in write operations) as before. The bus error signal must be removed before the halt signal is removed. The exception to this is the read-modify-write cycle which is never re-run as this would defeat its indivisible mode of operation.

If the processor receives a bus error signal and initiates exception processing, it attempts to place a number of data words on the system stack before executing an area of code designed to handle the exception. If another bus error occurs before the first instruction in this code is executed, the processor halts.

2.1.7 M68000 Exception Processing

The M68000 is always in one of three states. Its normal state is when it is sequentially executing instructions and referencing memory to fetch instructions and data and to store results. The one exception is the execution of the **STOP** instruction upon which the processor ceases to access memory until an external signal is received.

If, in its operation, the processor encounters what appears to be an irremedial hardware failure, such as a bus error occurring while processing another bus error, it enters its halted state. Once the processor is halted, only an external reset can restart it.

The third state which the processor can assume is the exception processing state. The exception may be generated as the normal result of some instructions (traps) or as the result of an abnormal condition arising during the execution of an instruction (e.g. an attempt to use a zero operand as a divisor). Exceptions are also generated as a result of hardware interrupts or signals. The reset condition is a special case of an externally generated exception.

2.1.7.1 Privilege States

The M68000 operates in one of two privilege states. The current privilege state determines which instructions may be executed, which stack pointer is active and may also be used by external memory management circuitry to determine which area of memory may be accessed. The current state of the processor is determined by bit 13 of the status register. When this bit is set, the processor is operating in the supervisor privilege state.

The supervisor state is the higher privilege state and all bus cycles generated by the processor while in this state are classed as supervisor references. All instructions which use address register A7 in this state access the supervisor stack pointer.

All exception processing is done in supervisor state regardless of the state of bit 13 of the status register. All data which are stacked during exception processing therefore uses the supervisor stack.

The lower state of privilege is termed the user state. Programmes executing in this state use a restricted instruction set as they are not allowed to perform system functions (e.g. to reset external devices). To ensure that any programme only enters the supervisor state in a controlled manner, instructions which access the entire status register are privileged instructions. The lower byte of the status register may be accessed in either state.

Programmes executing in the user privilege state use the user stack pointer. As an aid to programming, however, two privileged instructions exist which allow programmes executing in the supervisor state to access the user stack pointer as well. All bus cycles generated while the processor is in this state are classified as user references.

As the function code outputs distinguish both between user and supervisor references and between programme and data references, the use of external memory management circuitry can increase the total addressing range of the M68000 to 64Mb.

When the processor is reset, it begins execution in the supervisor state. In order to change to the user state bit 13 of the status register must be cleared. This can be done by any instruction which accesses this bit individually, or which accesses the whole of the status word. Once in the user state, only exception processing can cause a return to supervisor state.

2.1.7.2 Exception Vectors

When an exception occurs, the processor first saves a copy of the current status register. The status register is then set for exception processing (by setting the supervisor bit) and the exception vector number is determined. The exception vector number is an 8 bit operand which, when multiplied by four, gives the address of the exception vector. The exception vector itself is the address of a routine which will handle the exception.

The vector number may be determined in a number of ways depending on the type of exception. Some types of exception (e.g. bus error) have fixed vector numbers and these are therefore directly determined. Exceptions which are caused by **TRAP** instructions supply the vector number as a part of the instruction word. When an exception is caused by a hardware interrupt, the vector number is ususally supplied by the external device on bits 0-7 of the data bus during the interrupt acknowledge cycle. Where the device is incapable of doing this, the vector number is determined by the level of the interrupt.

The M68000 recognises 256 vector numbers (although some of these are reserved) and the exception vector table therefore occupies the bottom kilobyte of memory. This table is shown in Fig. 2.2.

2.1.7.3 Context Switching

After the processor has determined the exception vector, it saves the current context on the supervisor stack. For most types of exception the context consists of the status register and the programme counter (6 bytes). The saved programme counter points to the instruction which would have been executed next, had the exception not occurred.

To facilitate recovery after bus errors and address errors, an additional 8 bytes of contextual information is saved including the processor's instruction register and the address of the memory location which the processor was attempting to access at the time of the exception. In addition, the saved value of the programme counter is unpredictable and may point anywhere between the address of the start of the instruction which caused the error and the start of the following instruction.

When the processor receives a reset exception, no contextual information is saved at all.

After saving its previous context, the processor loads the programme counter with the value obtained from the exception vector and resumes normal operation.

2.1.7.4 Exception Priority

The interval between the receipt of an exception condition and its acknowledgement depends on the type of exception and the processor's current activity. There are three groups of exceptions. The reset, bus

error and address error exceptions have the highest priority (in that order) and cause exception processing to begin at the next bus cycle. The next group of exceptions are those caused by trace conditions, hardware interrupts, illegal instructions and privilege violations (in that order of priority). These exceptions are processed before the next instruction begins. (In the case of illegal instructions and privilege violations, the potential exception condition is recognised when the instruction is decoded and the exception processing is therefore initiated before the instruction is executed). The lowest priority group of exceptions are those which occur during the normal processing of an instruction. Exception processing is done as and when the exception occurs. There is no priority grouping within this group of exceptions.

When two exceptions occur simultaneously, the highest priority exception is processed first. When an exception occurs during the processing of another exception, the processing of the current exception is suspended if the later exception has a higher priority.

Hardware interrupts also have hierarchical recognition. Seven levels of interrupt priority exist and the recognition of an interrupt depends on the state of the interrupt mask in the status register. This mask is composed of bits 8-10 of the status register.

An interrupt request is generated by encoding the priority of the external device on the three interrupt request pins (see above). A zero on these lines indicate no pending request. The priority level of the incoming interrupt is compared with the setting of the interrupt request mask. If the priority of the interrupt is greater than the priority encoded in the mask, the mask is set to the level of the incoming interrupt and the interrupt is allowed. In this way, low level interrupts cannot interfere with the processing of a higher level interrupt. As the

status register is stored in the exception processing sequence and restored upon the execution of a return from exception (RTE) instruction, lower level interrupts always have a chance of recognition after the handling of a higher level interrupt.

A mask level of zero means that all interrupts are allowed. An interrupt level of seven is always allowed, whatever the setting of the interrupt mask.

2.1.7.5 Bus Errors and Address Errors

Bus Errors and address errors are in the highest priority group of exceptions. Apart from the reset exception which does not attempt to save any contextual information, they are the only types of exceptions which cause exception processing to begin before the completion of the current instruction. For this reason, recovery from such exceptions is somewhat difficult as the saved value of the programme counter is unpredictable.

The M68000 attempts to alleviate the recovery problem by saving more information on the system stack than for other types of exception. The extra information includes the instruction register at the time of the exception, the access address causing the exception, and a special status word giving the active function code at the time of the exception, the state of the read-write line and whether the processor was processing an instruction or not at the time of the error.

In general, this is insufficient information to allow a full recovery from the error, but it does allow the error handler to determine the cause of the error and may allow it to modify the internal registers so that the instruction can be re-run after the return from the handler. There is no instruction which specifies that the return from the

exception should expect a larger than normal stack frame nor is there any information contained in the stack frame which the processor can interrogate in order to determine the size of the frame. It is therefore the responsibility of the handler routine to clean up the stack frame and to determine where to continue normal execution.

Because full recovery after a bus error cannot be guaranteed, the M68000 is unsuitable for paged virtual memory applications which would generate frequent bus errors as a normal part of their operation.

2.2 The M68010 Microprocessor

2.2.1 General Features

The M68010[29] is the third processor in the M68000 range. The chip is pin-compatible with the M68000 and object code for the M68000 runs on the M68010. Due to increased functionality, the M68010 has a slightly larger instruction set than the M68000 and object code for the M68010 may generate illegal instruction traps when run on the M68000.

In general the operation of the M68010 is similar to that of the M68000, except that instruction execution is generally faster on the M68010. Differences between the two machines are highlighted below.

2.2.2 Register Set

The M68010 contains 3 registers in addition to those contained in the M68000. The vector base register is a 32 bit register which defines the start of the exception vector table. After the address of the exception vector has been determined as in the M68000, the value contained in the vector base register is added to it before the vector is fetched. In this way, the vector table is not constrained to be located

at the bottom of memory, and the opportunity exists to switch between a number of vector tables.

The two alternate function code registers are 3 bytes wide and, along with the privileged "move alternate address space" (**MOVES**) instruction, allow routines easy access to areas of memory which could normally only be accessed when executing in a different mode.

2.2.3 The M68010 Instruction Set

The M68010 instruction set contains instructions to access its extra registers. In addition, the **MOVES** instruction allows operands to be transferred between registers and memory using specified function code outputs rather than the function codes which would normally be output for the operation. The function codes used for such operations are stored in the alternate function code registers. When the transfer is from register to memory, the value in the destination function code register is used in the write cycle. When the transfer is from memory to register, the value in the source function code register is used in a read cycle. These outputs can be used by external memory management circuitry to direct access to particular address spaces.

2.2.4 M68010 Exception Processing

The exception processing sequence of the M68010 is basically the same as that of the M68000 but the format of the exception stack frame is different. The stack frame for all exceptions is at least four words in length and the fourth word in the frame (the word which is offset by 6 bytes from the value in the system stack pointer) contains in bits 12-15 a format code which specifies the length of the remainder of the frame. Bits 0-11 of this word contains the offset of the current

exception vector from the base of the vector table. Generic exception handlers can therefore determine the source of a particular exception. The M68010 recognises only two format codes. The first code is used by all types of exception except bus error and address error and indicates that the stack frame is 4 words in length. The second code is used for bus errors and address errors and indicate that the frame is 29 words in length.

This format code is inspected during the **RTE** (return from exception) instruction and is used by the processor to correctly remove the stack frame on resumption of normal processing.

2.2.5 Bus Error Handling

When a bus error occurs, 29 words of contextual information are stored on the system stack. This information is sufficient to allow the processor to continue the faulted instruction after return from the exception by only re-running the bus cycle in which the fault occurred rather than having to re-run the entire instruction. (In the case of a read-modify-write cycle, the entire instruction is re-run to maintain the integrity of the operation).

The user can opt to perform a software re-run of the instruction and in this case, the processor is informed by setting bit 15 of the fifth word in the stack frame. If this bit is set when an **RTE** instruction is executed, The processor still reads all the information from the stack frame but instruction execution continues as normal after the return from an exception.

This feature of instruction continuation after a bus error allows the M68010 to be used in paged programme applications. When a bus error occurs due to a page fault, the supervisor task would install the required

information in the correct place in memory and return from the exception whereupon instruction execution would continue as normal. The page fault would be transparent to the task.

2.2.6 Address Error Handling

Address errors are handled by the processor in the same way as bus errors. Care must be taken, however, if instruction continuation is to be used on return from the exception handler. If the information on the stack frame and relevant internal registers is not modified by the handler, the processor, on restoring its internal state will again try to access the same (invalid) address, causing another address error.

2.3 The Motorola M68451 Memory Management Unit

The M68451 memory management unit (MMU) comes in a 64 pin DIL package and was designed to work with the M68000 series of processors. It will be clear from the description of its features that it was designed to take advantage of many of the enhancements which exist on the M68010 processor, and the final hardware configuration used this combination reflects this. More than one M68451 are capable of being combined in in a system to provide more power and flexibilty. Logically, the operation of a multi-MMU system is the same as that of a single MMU system although there are physical dissimilarities. The operation of the M68451 is described here as if there were only one MMU in the system.

The M68451 partitions the logical address space into contiguous segments which are 256 bytes or larger. An address translation is performed for each segment in order to produce a physical address. Each segment may be defined as user, supervisor, programme or data (or some combination of these). In addition, any segment may be write protected.

When the bus master generates a logical address which the MMU cannot translate into a physical address, a bus error condition is generated.

Each bus master in a M68000 system must provide a 3 byte function code for every bus cycle which it initiates to specify which area of memory (address space) is to be used for that cycle. (These function codes appear on the output lines FC0-FC2 of the M68000 and the M68010 as described above). The M68451 uses this function code in its address translation. In addition, it provides a fourth input (FC3) to distinguish between bus cycles generated by the processor and cycles generated by some other bus master. This line would be connected to the **bus grant acknowledge** line of the processor.

The function code indexes an internal table of the MMU which contains an **address space number** for each defined function code. The resultant address space number is compared with the address space number of each defined segment in an attempt to find a match.

Segments are defined in a group of internal registers called descriptors. Each MMU has 32 descriptors and each descriptor defines one segment. The inclusion of more than one MMU therefore increases the number of segments that can be defined. Each descriptor contains 6 fields which may be independently set. These fields are described below.

2.3.1 The Logical and Physical Base Address Fields

These fields are both 16 bits wide and correspond to bits 8-23 of the input and output addresses. The logical base address specifies the base address of the segment which is defined by the descriptor. This base address will be mapped into a physical segment whose base is specified by the physical base address. The size of this segment is determined by the logical address mask.

2.3.2 The Logical Address Mask

The logical address mask is 16 bits wide and is used to determine which bits of the logical base address are significant in defining the segment by performing a bitwise logical AND operation. Provided the address space numbers match, a logical address is considered to lie within the segment if the result of AND'ing the upper 16 bits of the address with the logical address mask is the same as the result of AND'ing the logical base address with the logical address mask. The upper 16 bytes of the physical address is then generated by AND'ing the physical base address with the logical address mask. The low order 8 bits of the address remain unchanged. This method of address translation implies that the size of a segment is related to the address boundary which is its base address and further that the logical and physical addresses must lie on the same type of boundary.

2.3.3 The Address Space Number and Address Space Mask

Both of these fields are 8 bits and together define a group of address space numbers for which the segment is valid. When the relevant address space number for a bus cycle is ascertained from the address space table, the MMU searches its descriptors to find one whose address space number when AND'ed with its address space mask is the same as the cycle's address space number when AND'ed with the address space mask. When such a segment is found, the addresses are compared as described above. If no segment can be found with both a valid address space number and a valid address range, a bus error is generated.

2.3.4 The Segment Status Register

This field is 8 bits wide and describes some general features of the segment. If bit 0 is set, the segment is valid. Only valid segments are used in address translations. If bit 1 is set, the segment is write protected. A write cycle within such a segment will generate a bus error condition. Bit 7 is set by the MMU if the segment has been used in a successful translation since it was defined, whereas bit 2 is set if the segment has been written to since its definition. These two bits facilitate the implementation of virtual memory systems. If bit 4 is set, the MMU will interrupt the processor if the segment is accessed and set bit 3 to indicate which segment caused the interrupt. Such a facility provides an alternate signal to the bus error which is generated by a failure to translate the logical address. The remaining bits of the segment status register are undefined.

2.3.5 Global Registers

In addition to the 32 descriptors and the address space table, the M68451 contains seven global registers. The accumulator is a register in the same format of the descriptors which can be accessed by the processor. Data destined for a descriptor is first loaded into this register and the number of the destination descriptor is loaded into the 8 bit descriptor pointer. The descriptor is loaded by initiating a load descriptor operation at which stage the MMU checks that no descriptor which is already loaded and enabled attempts to translate the same logical address in the same address space. If such a descriptor is found, the operation fails and an error is signalled. If no such descriptor is found, the destination descriptor is loaded with the data in the accumulator. The segment status register is the only field of a

descriptor which may be read from or written to independently of the other fields in the descriptor.

The M68451 contains a local and a global status register. The local status register reflects the outcome of the previous MMU operation while the global status register is useful in a multi-MMU system as it provides an indication of which MMU was responsible for the previous fault condition. The global status register also determines whether an interrupt generated by a descriptor is actually passed to the processor.

Two additional descriptor pointers provide indications to the last active descriptor. The result descriptor pointer contains the descriptor number of the last descriptor to be loaded or to be used in a translation. The interrupt descriptor pointer contains the number of the last descriptor to request an interrupt. The interrupt vector used by the MMU is held in the 8 bit interrupt vector register.

2.3.6 Priority and Timing

In address translation, descriptor 0 of an MMU has the highest priority and is searched first. Higher descriptor numbers have correspondingly lower priority. In a multi-MMU system, the units are chained and the first unit in the chain has the highest priority and is the first to attempt a translation. Priority decreases down the chain.

The duration of any MMU operation is dependent on the state of various signal lines at the time the operation is begun. In a single MMU system, the translation during a read cycle takes between 11 and 13 clock cycles while a write cycle takes between 9 and 11 clock cycles. To load a descriptor takes 17-35 cycles depending on error conditions. A desciptor can be disabled by writing to its segment status register. This takes 11-13 cycles. The M68451 is also capable of translating an

address and putting the result in its accumulator for use by the processor. Such an operation takes 19-29 clock cycles in a single MMU system. Most operations other than read and write cycles are slower in a multi-MMU system.

2.4 The Hitachi HD68450 Direct Memory Access Controller

The HD68450 is a four channel direct memory access controller (DMAC) whose channels each operate independently of the others and which comes in a 64 pin DIL package with signals which are directly compatible with those of the M68000 bus and those of the M68451 memory management unit. The DMAC functions by transferring a series of operands between memory and an external storage device quickly, and with minimum intervention by the processor. Operands may be 8, 16 or 32 bits in length and the number of operands transferred in a single operation is variable. The HD68451 can also regard an area of memory as being a device and is therefore capable of high speed memory-to-memory transfers.

2.4.1 HD68450 Signal Description

The HD68450 contains a full set of signals for M68000 bus operation. The data bus and bits 8-23 of the address bus are multiplexed onto 16 pins and are de-multiplexed by two signal lines and external logic. Bits 1-7 of the address bus are not multiplexed. Three function code lines FC0-FC2 perform the same function as the function code outputs of the M68000, ensuring that correct address translation is performed for bus cycles in which the HD68450 is bus master.

In addition to such signals which are used by all channels of the DMAC, each channel has 3 control lines which affect that channel only. The channel request line is an input to the DMAC and indicates that the

device associated with the channel is ready to begin a transfer operation. The channel acknowledge line is asserted when the M68000 bus has been acquired for the channel and the bus cycle is beginning. The peripheral control line is a multi-purpose line which can be programmed to act as an output line to provide a START signal to the device or as an input line which is asserted on various status conditions of the device.

2.4.2 HD68450 Register Description

The HD68450 contains 69 internal registers of which only one (the general control register) is common to all channels. The remaining 68 registers are divided into four identical sets of 17 registers and each set controls the operation of one channel. Only four bits of the eight bit general control register are defined and these determine the portion of the bus bandwidth which the DMAC uses when transferring data at a limited rate

Each channel contains three 32 bit address registers. The memory address register contains the address of the location of the operation in memory and may be automatically incremented or decremented during a transfer. The device address register contains the address of the data port of the device which is associated with the channel and may also be incremented or decremented. The base address register contains a reference address for "continued" or "chained" operations, which are described below.

The HD68450 has two 16 bit transfer count registers per channel. The memory transfer counter contains the number of operands which are remaining to be transferred in the current operation. This register is decremented as each operand is transferred and the operation is complete when this register contains 0. As the transfer counter is 16 bits wide,

a maximum of $2^{16}-4$ bytes may be transferred in a single operation. It is possible to "chain" operations, however, to provide unlimited transfers. The base transfer counter is used in such chaining operations which are described below.

Three 8 bit registers specify the function codes which are to be output during bus cycles. The memory function code register holds the function code which is used when the address on the address bus is generated from the memory address register while the device function code register contains the code for cycles which are governed by the device address register. The base function code register is used to refresh the memory function code register in "continued" operations described below.

Two interrupt vector registers are provided per channel. At the end of a successful operation the DMAC will interrupt the processor to signal completion. The vector number for this operation is contained in the normal interrupt vector register. If an error condition arises during a transfer, the DMAC will abort the transfer and interrupt the processor, providing the contents of the error interrupt vector register as the vector number.

Four 8 bit control registers define the mode in which the DMAC operates. The device control register specifies the size of the device's data port (8 or 16 bits), the type of device which is attached to the channel (M6800 compatible, M68000 compatible or having a known signal configuration), the mode in which bus cycles are requested (described below) and the operation of the programmable peripheral control line. The operation control register defines the size of the operand (8, 16, or 32 bits), the direction of the transfer, whether the operation is simple, continued or chained, and the conditions which initiate the transfer. The sequence control register defines whether the memory and device registers

are (independently) incremented, decremented or unchanged during a transfer. If both registers are programmed to change and the operation control register programmed to provide internal initiation of the transfer, the DMAC can be used to provide memory to memory transfers.

The final control register is the channel control register which is used to start, abort or continue the transfer and to enable or disable interrupts from the channel. This register may also be read to determine the state of the current operation.

The channel error register contains an error code if an error is indicated by the channel status register. If no error is indicated, the contents of the error register are undefined. The channel status register also contains information about the state of the last or current transfer and the state of the peripheral control line. Both registers are 8 bits wide.

The operation of the four channels is prioritised and the priority of the channel (0-3) is held in the 8 bit channel priority register. In operation, it is advisable to ensure that all channels are given a different priority.

2.4.3 Bus Utilisation

The HD68450 is able to request and relinquish the M68000 bus under a variety of conditions. If the operation control register is set for "auto-request" operation at the maximum rate, the DMAC requests the bus when the operation is initiated and retains the bus until the transfer is complete. If the operation is set to proceed at a limited rate, the DMAC monitors the bus activity in equal length sample intervals. If the activity was low in the previous sample period, the DMAC will request the bus and retain it for a portion of the next sample interval. If the

transfer is not completed in this time, the process is repeated for further sample periods. The length of the sample period and the proportion of this in which the bus is retained are defined by the general control register.

The DMAC can also be programmed to request the bus under the control of the device attached to the channel. In this case, the DMAC requests the bus when the device's request line is asserted. If the device control register is set for "burst" operation, the DMAC will retain the bus until the transfer is completed or aborted. If in this time, the device pauses in the transfer, the DMAC will allow one of its other channels to use the bus. If the request signal is negated before the first transfer has started, the DMAC will terminate the cycle and relinquish the bus.

The device control register can also be programmed to provide a "cycle-steal" mode of operation. In this mode, when the device pauses during a transfer, the DMAC will attempt to service other channels, but, if no channel requires service the DMAC will relinquish the bus unless the "hold" mode is also specified. If this is the case, the DMAC will monitor the request line in the same way as bus activity is monitored in an auto-request transfer and will relinquish the bus if the activity is low.

A hybrid mode of operation is available in which the DMAC auto-requests the first transfer of an operation but subsequent requests are initiated by the device.

To transfer data to or from a device, the data port on that device must be addressed by the DMAC. Some devices, however, have signals which, when asserted indicate that the present data on the data bus should be latched into its data port (write cycle) or that the device

should put data onto the data bus (read cycle). If such signals are used, the DMAC need not explicitly address the device (by loading its address from the device address register onto the data bus). If such signals are not used, the DMAC must address the memory location and the device data port explicitly in two separate cycles. The data is then held in an internal holding register between the two halves of the operation. Such transfers therefore use more of the bus bandwidth.

2.4.4 Complex Operations

The amount of data which can be transferred in a simple operation is limited by the size of the memory transfer counter. In addition, the data transferred in a simple operation is constrained to lie in a contiguous block of memory. Three ways of overcoming these limitations are available with the HD68450 DMAC.

If bit 6 of the channel control register is set when the transfer of a block completes, the memory address register, memory transfer counter and memory function code register are loaded from the base address register, base transfer counter and the base function code register respectively and the transfer continues using the new values. The contents of these registers and bit 6 of the channel control register must therefore be set by the processor before the transfer terminates.

Another method of increasing the size of a transfer but which removes the necessity for processor intervention in the middle of a transfer is array chaining. Using this method, a contiguous block of memory is initialised by the processor to describe the entire transfer. The block of memory is divided into a number of 6 byte structures of which the first four bytes contain the memory address of a block of data, and the remaining two bytes contain the number of operands in the block.

The DMAC is initialised with its base address register containing the address of the first 6 byte entry in the descriptive array, its base transfer counter containing the number of 6 byte structures in the array, and its operation control register set to indicate that the transfer is array chained.

When the transfer is started, the memory address register and the memory transfer counter are loaded from the values in the array. The base address register is incremented by 6 to point to the next structure in the array, and the base transfer counter is decremented. A simple block transfer then takes place as described above. At the end of the block transfer, if the base transfer counter is not zero, the internal registers are intialised with the new data pointed to by the base address register and the operation is repeated. If the base transfer counter is zero, the DMAC signals that the transfer is complete.

When array chaining is used, the descriptive structures must all be contiguous in memory. This approach uses minimum space but adding and deleting elements involves a time consuming copy of at least a portion of the array. In addition, the number of structures is limited by the size of the base transfer counter (16 bits). Another method of chaining simple operations is available which removes these limitations at the expense of an increase of memory usage. This method is called linked chaining.

The descriptive structures for a linked operation are 10 bytes long and need not be contiguous or ordered in memory. The first 6 bytes of the structure are the same as for array chaining and the final 4 bytes contain the address of the next element in the sequence. The DMAC is initialised with its base address register containing the address of the

first element and its operation control register set to indicate that the transfer is link chained. The base transfer counter is not used.

When the memory address register and memory transfer counter are initialised, the base address register is loaded from the last four bytes in the structure, so that it points to the next element. The transfer is complete when a zero pointer is encountered.

2.4.5 Operation Timing

The HD68450 bus cycles are essentially the same as those described for the M68000 above. The rate of transfer of data is limited both by the memory response times and the device response times. In addition, the transfer rate of any one channel is affected by its priority and the activity on the other channels. If the transfer is set to proceed at a limited rate then the transfer rate is affected by all other bus activity as well. Unless operating in burst mode, the DMAC will also be affected by the activities of other potential bus masters in the system as it may have to request the bus several times in a single transfer.

2.5 The Marksman T Series Winchester Disk and Interface

The Marksman T Series hardware consists of a processor interface or controller, a selection of hard disk drives with capacities ranging from 20Mb to 160Mb as well as tape drives. Each controller is capable of controlling up to two hard disks and four tape drives.

The controller contains a write-only command register and a read-only status register each appearing in the same memory location and each being 16 bytes, though not all byte positions are defined. The operations available for disk include writing or reading multiple data blocks, formatting, verifying, track seeking, and identification of drive capacity.

In addition, as the power-on and power-off procedures are lengthy and involved, commands are also included to sequence the drive up after power-on and sequence the drive down before power-off.

To write or read a number of contiguous blocks from the disk the drive number, cylinder number, head number, first sector number and the number of sectors to transfer must all be loaded into the command register along with the read or write command. Data is buffered into 2 1kb FIFO's which are automatically swapped when full or empty. A signal is provided for use in systems with direct memory access controllers, to allow the transfer to be unbuffered and proceed at the maximum rate. At the end of the operation the status register contains information regarding the success or failure of the operation as well as information about the state of the drive and the position of the heads.

When a tape drive is used, some operations such as rewinding may be carried out concurrently with disk operations. Other tape commands include reading and writing blocks and writing and searching for file marks. In addition the contents of the disk drive may be transferred directly to tape on a per-cylinder basis using the backup and restore commands.

The 20Mb and 40Mb disk drives have average head positioning times of 65ms and maximum positioning times of 130ms. The average and maximum positioning times on the 80Mb and 160Mb models are 50ms and 100ms respectively. All models rotate at 2400rpm and have an average rotational latency of 12.5ms. The data transfer rate for the 160Mb drive is 1.28Mb/second and 960kb/second for all other models. Access to the command and status registers and to the FIFO are governed by the normal bus conditions.

2.6 The Adaptec ACB-4000 Series Winchester Disk Controller

The ACB-4000 controller boards interface Winchester disk drives to any Small Computer System Interface (SCSI) standard host adapter interface. The SCSI interface has a 9 bit data (including one parity bit) bus and 9 signal lines which indicate the state of the controller and the device. The physical layout of the disk is completely transparent to the host processor and logical sector addressing is used.

Commands are given to the device by transferring a "device control block" to the controller as it requests it, one byte at a time. Depending on the type of operation being performed, the device control block may be 6 bytes or 10 bytes in length. The information contained in the device control block varies according to the command being given, though certain fields are static. Normal disk commands are available.

The time taken to transfer a command to the controller or to read status messages from it is completely under the control of the controller which may, in theory, request data from the processor in no particular order.

2.7 The FD179X-02 and WD279X-02 Floppy Disk Controllers

The Western Digital Corporation's FD179X-02 and WD279X-02 families of floppy disk formatter/controller (FDC) chips are software compatible 40 pin chips. The WD279X-02 family is the successor of the FD179X-02 and features slightly enhanced timing and application scope. The two chips are not hardware compatible. The FDC's will control up to four 8" or 5¼" floppy disk drives for using single-sided or double-sided, single-density or double-density disks.

Both models have an 8 bit data bus and two address lines. Internally, they contain five 8 bit registers with the status and command

registers occupying the same memory location. The command register is a write-only register and is used to transfer commands to the controller. The status register is read-only and contains information about the state of the currently selected disk drive and the result of the previous command.

The track register should always contain the number of the track where the head is currently positioned. It is a read-write register and the user must therefore ensure that only the correct data is written into it (when changing disk drives, for example). If the position of the head is uncertain, the RESTORE command always returns the head to track 0 using a signal output, rather than the contents of the track register.

The sector register is written to indicate the desired track in a read or write sector operation. The data register is the location of all data read from or written to the disk. Apart from this register, all I/O is unbuffered. In a SEEK operation, the track number of the desired destination track is written to the data register before the operation is started.

The status register provides information whose format depends on the previous command. In general it indicates whether the drive is ready, busy or write-protected, the type of error (if any) in the execution of the command and the state of some of the drive control lines. External circuitry is required to provide an interface to the chip in order to easily perform drive, side and density selection and the enabling/disabling of the interrupt request and data request signals.

The commands available include reading and writing sectors and complete tracks, stepping the head in or out by a number of tracks, seeking to a specific track, moving to track 0 and forcing an interrupt request.

2.8 The Maxtor XT-1000 Series Winchester Disk Drives

These hard disk drives are available with capacities which approximate to 60Mb, 100Mb and 140Mb. All models rotate at 3600rpm and have an average rotational latency of 8.33ms. The movement of the head to an adjacent track takes 5ms and the average and maximum seek times are 30ms and 48 ms respectively.

2.9 The DRE-7200 Floppy Disk Drive

These floppy disk drives take 8" single or double-density floppy disks with one or two sides. Disks rotate at 360rpm with an average latency of 83ms. Loading the head takes 35ms including settling time, while moving the head to an adjacent track occupies 6ms. At the end of the seek operation, the settling time for the head is 14ms.

2.10 The M6850 and the SY6551 Asynchronous Communications Adapters

Both of these serial devices (ACIA's) were designed for the Motorola M6800 family of processors and the SY6551, designed by Synertek, may be regarded as an enhanced version of the M6850 (Motorola), although the chips are neither hardware compatible nor software compatible with each other.

2.10.1 The M6850 ACIA

The M6850 contains four 8 bit registers in two memory locations, each register being either read-only or write-only. The transmit and receive data registers occupy the same memory location. Data to be transmitted in serial form is written to the transmit data register while received serial data is buffered in the receive data register.

The control register is a write-only register which is divided into four fields. The first field is one bit and enables and disables the interrupt request generated on receipt of data. The second field occupies two bits and is used to set the sample ratios in the receiver and transmitter. Divide ratios of 1, 16 and 64 are available. In addition, if both bits are set high, the device is reset. Resetting does not affect the other fields of the control register.

The third field in the control register is three bits wide and controls the data format. Formats of 7 and 8 bit data are available with odd, even or no parity and one or two stop bits. The fourth field is two bits wide and controls the transmitter interrupt request signal. If both bits are set high, a break condition is transmitted. When a character is transmitted with the transmitter interrupt enabled, the interrupt request can only be cleared by writing another character into the transmit data buffer or disabling the interrupt. The processor must therefore always disable the transmitter interrupt at the end of a message.

As the control register is write-only, it is helpful to keep a copy of what is written to the register in memory, so that the state of the device may be ascertained.

The status register is a read-only register in the same location as the control register. Its eight bits indicate when the receive buffer is full or its transmit buffer is empty, whether the device has generated an interrupt request, whether an error has occurred and the state of two signal lines.

No internal provision is made for internal data rate setting so that, in practice, the M6850 operates at a fixed baud rate which is determined by external circuitry.

2.10.2 The SY6551 ACIA

The SY6551 overcomes some of the limitations of the M6850 at the expense of a limitation in the available data formats. The formats which are available are sufficient, however, for most applications. The SY6551 contains two control registers, both of which are read-write and has programmable baud rate settings for all the commonly used rates.

The status register gives the same information as the M6850's though the bit patterns are different. If the status register is written to, the device is reset.

2.11 The M68230 Parallel Interface/Timer

The M68230 parallel interface/timer (PI/T) was designed for M68000 systems and provides double-buffered parallel interfaces and an operating system oriented timer. Logically, the PI/T appears to have three parallel ports. Ports A and B are true multi-purpose double-buffered parallel interfaces whereas port C is a multiplexed port which can act as a single-buffered general-purpose parallel interface but which is normally used to provide control functions for the timer and ports A and B. In total, the PI/T contains 22 registers. The port C data register and port C data direction register are related to both the timer and parallel interface functions of the chip. Most other registers may be regarded as being related to only one of the functions. Functionally, the PI/T operates as two independent modules and will therefore be described as such.

2.11.1 The M68230 Parallel Interface

The parallel interface consists of two 8 bit ports A and B. Both ports are double-buffered and may operate as input, output or

bi-directional ports. The port general control register controls the overall operation of the interface and this register contains a 2 bit field which specifies the mode of operation. Some modes require further specification and a two bit field in the port control register for each port indicate the port's submode.

Four handshake lines (H1-H4) provide the external control for the port and its associated device if any. These lines perform different functions depending on the mode which the PI/T is operating in. It is therefore possible to tailor the operation of the PI/T to suit a large number of applications.

When operating in unidirectional modes, a data direction register associated with each port controls the direction of data for each pin. This is known as the primary direction and data transfers in this direction are double-buffered and controlled by the handshake pins. Single-buffered or unbuffered data paths exist in the opposite direction. When in bidirectional mode, the instantaneous data direction is determined by the state of the handshake pins. As data transfers may happen in any order and the state of the handshake pins depends on the external system, some external arbitration logic is usually required to control the data flow.

Both of the data buffers for each port are accessible at any time. The port status register always shows the instantaneous state of the handshake pins and this gives some indication of the flow of data. Some software arbitration is however usually required to determine which of the buffers is currently relevant.

Port C can be programmed to provide request signals for direct memory access transfers and interrupts to the processor. These signals are controlled by the state of the handshake pins and their operation

therefore depends on the mode of the interface. Interrupt requests may be vectored or autovectored. Bits 0 and 1 of port C have no special function either for the parallel interface or for the timer. These lines can therefore be used to provide extra control lines for the peripheral.

2.11.2 The M68230 Timer

The M68230 timer can generate periodic interrupts, a square wave, or a single interrupt after a programmed period. It can also be used for elapsed time measurement. The mode of the timer is controlled by the timer control register which also is used to enable timer and the interrupt request line.

The PI/T contains a 24 bit synchronous down counter which is loaded from three 8 bit registers. When this counter has counted to zero, bit 0 of the timer status register is set and an interrupt is generated if enabled. The interrupt request is negated when the status register is read. If the timer is programmed to provide a periodic interrupt, the counter is automatically reloaded at this stage.

Chapter 3

The Hardware Configuration

Research for this project was carried out using two computer systems designed at the University of Bath. The original system was a multi-board system which was marketed under the name of Darkstar and will be referred to by that name. The successor to the Darkstar computer was a single-board system which will be referred to by the mnemonic SBC.

Both computer systems used a selection of the hardware described in the previous chapter and the SBC system was designed so that circuit boards designed for the Darkstar system could be used in conjunction with one or more SBC's as an extension to the system with little or no modification.

3.1 The Darkstar Computer System

The design of Darkstar began in 1981 with the design and construction of the processor board. This board contained an M68000 microprocessor and one or two M68451 memory management units, but no direct memory access controller. This meant that, if direct memory access was desired, the DMAC had to be located further down the backplane than the MMU. Bus cycles generated by the DMAC did not, therefore, use the MMU so no address translation was available on such cycles. The processor board was later re-designed to include an on-board Hitachi HD68450 DMAC and it was this version of the processor board which was used in the implementation of the UNIX operating system.

The processor board in Darkstar occupied the first position on the backplane. The interrupt request lines were "daisy-chained" away from this position so that the highest priority interrupts were from boards

nearest the processor board. Boards which did not use the daisy chain had links made which connected the daisy chain through to the next board on the backplane. Slots which were empty, however, needed to be wired on the backplane in order to continue the daisy chain.

DMAC per-channel signal lines also had to be hardwired on the backplane thus associating a channel with a particular backplane slot, rather than with a particular device. This, along with the daisy chain requirements meant that a the order of boards along the backplane, once decided, was, to a large extent, fixed.

Contemporary with the design of the processor board was the design of a ¼Mb relocatable memory board with full error detection and correction. Later memory boards had capacities of ½Mb, 1Mb and 2Mb with error detection. The first implementation of the UNIX operating system on Darkstar ran with one ¼Mb board, but it was found that, for reasonable speed of operation with one or two users, at least three such boards were required in the system. Later Darkstar systems supported up to eight simultaneous operating system users and it was found that 2Mb was the minimum memory requirement for acceptable operation.

Two hard disk interface boards were designed for Darkstar. The first contained a Marksman T Series interface and 40Mb and 160Mb disks were used in UNIX systems. The more common disk interface, however, used the Adaptec ACB-4000 series Winchester disk controllers. The interface board incorporated an 8 bit read-write register which allowed the user to set and inspect the state of the SCSI bus signals. The Maxtor XT-1000 series Winchester disks (or other SCSI compatible disks) were used with this interface board. Eight inch floppy disk drives were also supported on Darkstar systems, using an interface board containing one of the the FD179X-02 series controllers.

Serial I/O was available on Darkstar via the M6850 ACIA's. Two ACIA's were contained on a board which also contained drivers for diagnostic light-emitting diodes. Later, another I/O board was added which contained two INS8250 ACIA's (not mentioned above) as well as a parallel interface, 1kb of battery-backed RAM and a real-time clock. More serial ports were made available by a board which used a Motorola interface chip which controlled 8 M6850 ACIA's.

Non-volatile data could be stored in EPROM's and the EPROM board designed for Darkstar contained up to 16 2716 EPROM's in 8 pairs. One EPROM in the pair contained the upper data byte while the other EPROM contained the lower byte. Each EPROM card therefore stored between 2kb and 16kb of data.

The basic Darkstar system for the UNIX operating system therefore contained the following hardware:

a) A processor board containing an M68000 processor (M68010 for the UNIX System V system), two M68451 memory management units and a HD68450 direct memory access controller.

b) Memory boards of differing capacities but giving in total at least ¾Mb of random access memory.

c) Two serial ports.

c) An SCSI interface board controlling a Winchester disk with a capacity of at least 40Mb.

d) A floppy disk interface board and one or two 8" floppy disk drives.

e) An EPROM card with at least 2 2716 EPROM's to contain the UNIX first-level bootstrap programme.

The Darkstar memory map is shown in Fig 3.1.

3.2 The Single Board Computer (SBC) Configuration

The design of the single-board computer began in 1983 when the need for increased processing power over that provided by Darkstar was recognised. In contrast to Darkstar, the SBC was designed for the UNIX system, although the need to maintain compatibility with Darkstar hardware necessitated the retention of some features of Darkstar which were incompatible with the UNIX environment.

The other design criteria for the SBC were that it should operate in a stand alone mode as a complete computer or as a controller card and that several SBC's should also be able to fitted to a common backplane to produce a multi-processor system with shared memory and inter-processor signalling.

3.2.1 Standalone Configuration Features

The SBC contains an M68010 processor, a HD68450 direct memory access controller, and one or two M68451 memory management units. Bus cycles generated by the DMAC use the MMU's so that full address translation is available on all bus cycles. Because of the method of on-board bus arbitration used, the DMAC must be used in dual-addressing mode when accessing the two local devices (hard disk and floppy disk).

Each SBC contains some on-board dynamic random access memory which can be in one of the following three configurations:

a) 32 64Kbit by 1 bit chips providing a total of ¼Mb of RAM.

b) 16 256Kbit by 1 bit chips providing a total of ½Mb of RAM.

a) 32 256Kbit by 1 bit chips providing a total of 1Mb of RAM.

The required option is selected by changing the address decode logic which is resident in two PAL's. The size of on-board memory affects the values which the processor number (described below) may take. The

ability to store up to 8Kb of non-volatile data is provided by two 2732 EPROM chips.

Serial I/O is provided by two SY6551 ACIA's while an M68230 PI/T provides both a parallel I/O interface and the timer functions. Mass data storage may be either on floppy disks or hard disks. The floppy disk controller is one of the WD279X-02 family with two additional read/write registers added. The first additional register is the same as the additional register on the Darkstar floppy disk interface board while the second register is used to store the interrupt vector. The floppy disk controller is connected to channel 0 of the DMAC.

Where a hard disk is desired, the PI/T is used as an SCSI interface to an Adaptec ACB-4000 controller. In this mode, port B of the PI/T interfaces the SCSI data bus and port A interfaces those signal lines which are frequently used. The reset and select signal lines are connected to the unallocated bits (0 and 2) of port C. The port interrupt request, interrupt acknowledge and DMA request lines then perform those functions for the Winchester disk controller. It can be seen, therefore, that, when a Winchester disk is connected to the SBC via its on-board interface, no other form of parallel I/O is available using the PI/T. The external logic required to interface the SCSI signals to the PI/T handshake lines is resident in a PAL.

The PI/T is connected to channel 1 of the DMAC. The remaining two DMAC channels are free and may be used to control external devices or for memory-to-memory transfers. In particular, in a multi-board system, one or both of the spare DMAC channels may be used to perform inter-processor memory copies, thus reducing the CPU time and the global backplane bandwidth needed for inter-processor communication.

The memory map of the SBC is given in Fig. 3.2 while the interrupt priority levels of all local devices are listed in Fig. 3.3. The non-maskable abort switch is local to a particular SBC as is a local reset switch. Provision is also made for a global reset signal via the backplane.

The schematic diagram of the SBC is shown in Fig. 3.4 and a photograph of the actual board is given in Fig. 3.5.

3.2.2 Multi-Processor Configuration Features

Each SBC contains an 8 bit local register of which the lower 6 bits are used to hold the processor number. The contents of this register are set by a group of on-board switches and is therefore read-only under normal operation. In a multi-processor system, each processor must have a unique processor number.

The processor number determines where, in the global memory map, the local memory for each processor appears. Each processor accesses its own local memory from locations starting at 0 and going up to 40000_{16}, 80000_{16}, or 100000_{16} depending on the amount of memory it contains. These accesses do not use the backplane. In addition, each processor can access the local memory of all processors in the system (including its own) via the backplane as a contiguous block of memory beginning at $40000_{16}\times$(processor number). Processors with ½Mb of local memory are constrained to having even processor numbers while the processor number of processors with 1Mb of local RAM must be a multiple of four.

Each SBC also contains an 8 bit **processor control register** which appears in the global memory map at location $840B81_{16}+(2\times$(processor number)). This is a read/write register of which

only the top four bits are used. Using this register, each processor can affect the operation of any processor in the system including itself.

If bit 5 of the processor control register is cleared, a reset exception is generated. This bit is automatically reset at the end of the exception processing sequence. Bit 7 of the control register reflects and controls the state of the halt signal to the processor. When this bit is cleared, the halt signal is asserted and the processor ceases execution of instructions. To negate the halt signal and resume normal instruction execution, bit 7 must be reset. In addition, if the processor halts due to some other condition, bit 7 is cleared. This bit is not automatically reset so, if the processor is reset after a halt condition and is reset without changing the status of bit 7, the processor will halt upon completion of the reset exception processing.

When bit 6 of the control register is cleared, the processor receives an autovectored interrupt request at priority level 5. This is the only inter-processor signalling mechanism which is available so software arbitration is necessary to determine the type of action to be taken on receipt of such a signal. Bit 6 is automatically reset when the interrupt is acknowledged.

When the processor is reset, it fetches 8 bytes of data from memory locations 0-7 and uses these to load the programme counter and stack pointer. Where a bootstrap programme is held in EPROM, the bottom 8 bytes of the EPROMS can be mapped into the bottom 8 bytes of memory so that the processor always uses a constant reset vector. However, this mapping is only useful during reset exception processing and, if it remains active during normal processing, locations 0-7 will always be read-only. It is therefore helpful to be able to disable the mapping when normal processing resumes and this action is controlled by bit 4 of

the processor control register. When this bit is set, the mapping is enabled. Clearing this bit disables the mapping but leaves the entire memory area of the EPROMS accessible at the normal address.

3.2.3 Multi-Processor Operation

In a multi-processor system, one SBC is designated as master and all other processors as slaves. These roles are determined by links on the board. The master processor is set so that, on reset, its halt line is negated and the bottom 8 bits of its EPROMS are mapped into the bottom 8 bits of memory.

All other processors are set to halt on reset with the EPROM mapping disabled. When the system is reset, therefore, only the master processor runs and executes a bootstrap programme stored in EPROMS while all other processors are halted until a signal is received from the master processor. The master processor may therefore load any programme into the slaves and remove the halt condition by resetting bit 7 of the processor control register.

Once running, processors may transfer data to and from each other by using the global memory mapping and signal each other by means of bit 6 of the processor control register. Each processor is, however, capable of halting or resetting any other processor so the master/slave relationship is no longer enforced by the hardware. Strict software control is therefore necessary to ensure that no processor accesses the control register of another processor in an illegal manner.

Any processor in the system may be set to respond to interrupts generated by external devices connected to the global backplane, but it is clear that, for normal interrupt processing to take place, only one processor may be configured in this way.

Accesses to the backplane are controlled by an arbiter board, one of which is necessary for any multi-processor system. When a processor initiates a bus cycle which attempts to access an off-board address, the backplane is requested on one of four request lines. Each of these lines is daisy-chained, so that any number of processors is allowed. The backplane is granted to the first requesting processor on the highest priority daisy-chain which responds by asserting an acknowledge signal.

During the period of arbitration, the halt signal to the processor is asserted. If the backplane request fails, a bus error is signalled while the halt line is still asserted. The processor will then attempt to re-run the bus cycle as described above. As no attempt is made to re-run the **TAS** instruction, bus error exceptions may be generated when trying to access valid off-board memory locations when using this instruction. As this instruction is crucial to the synchronisation of multi-processor operations, additional software arbitration is needed to ensure the correct operation of this instruction.

Chapter 4

The Operating System

4.1 The Choice of the UNIX System as the Target Operating System

The factors affecting the choice of the UNIX[23] operating system (developed by Bell Laboratories) as the target operating system for this project are discussed below.

4.1.1 The Advantages of the UNIX System

Since the early 1960's an increasing number of multi-tasking operating systems have been developed and have become generally available. Although they can be grouped into a few general categories, each operating system presents a slightly different interface both to the user(s) and to the hardware.

The routines which make up an operating system can be divided into three very broad categories: (a) the kernel, (b) the superstructure and (c) applications programmes. The kernel consists of those routines which interface directly to the hardware and provide such facilities as processor and memory allocation, I/O control and file management. The superstructure provides the interface between the user and the kernel and consists of routines like a Command Line Interpreter (CLI) which allow the user to communicate with the kernel without necessarily being aware of its operation. Applications programmes are facilities which are provided to perform generally useful functions such as text editing, sorting or word processing.

One group of operating systems (systems like RSX-11M, developed by Digital Equipment Corporation) are characterised by having a large proportion of kernel routines to superstructure routines. This means that

the range of facilities which are provided in the user interface is extremely rigid and difficult to change. User processes are often denied access to an arbitrary group of general-purpose routines. The UNIX operating system, on the other hand, has an extremely wide and flexible user interface which gives the user the opportunity to incorporate '*system calls*' into his programmes. For example, the UNIX CLI (the '*shell*') can be considered as an applications programme as it is possible to replace the standard shell by any other CLI which the user may require in order to provide a different command syntax or a modified set of facilities. In the context of this project, this would allow a special CLI to be written to provide a simple interface to the special features of the operating system which are concerned with real-time control, while retaining the standard shell for ordinary users, if this were desired.

Another feature of the kernel/superstructure division in the UNIX system is that it provides a uniform interface to all types of files and devices. In an operating system like CP/M[tm] (developed by Digital Research Inc.) the filing system is closely related to the characteristics of the storage medium and this imposes on extra degree of device dependency on the system, making it more difficult to support a large or changing selection of storage devices.

To the user of the UNIX system all files and all devices appear in the same format. Thus, a stream of characters can be sent to a named file on a floppy disk or to teletype terminal using the same procedure. The UNIX kernel determines the type or device with which the user wishes to communicate and acts accordingly, but this is completely transparent to the user. At the hardware interface, device drivers also have no need to know the structure of the filing system. When using a block structured device like a floppy disc, for example, a logical block number

is passed to the driver and the writer of the driver is free to translate this in any (consistent) way he likes into a physical block on the disc. The UNIX system therefore provides a very flexible interface both to hardware and to users.

Because of the flexible hardware interface, the UNIX system is easily portable across a wide range of computer systems, even though it was first designed for the PDP-7 and PDP-9 range of computers. It requires as its minimum hardware configuration a CPU, about 256kb of RAM, a simple memory management unit, an interrupting timer, a disc drive offering about 10Mb of storage and a terminal connected to a serial interface. In this respect it differs strongly from operating systems like OS/360 (developed by International Business Machines) which were designed specifically for the architecture of particular machines (in this case the IBM/360 and the IBM/370).

Closely related to the goal of flexibility is the goal of simplicity. To this end, present versions of the kernel contains less than 10% of assembly language code, with the remainder written in the high level language C. This is not only an aid to portability but also to system development as hardware drivers can be written in a form which makes them easier to modify and to debug.

Many of the concepts of the UNIX system also show evidence of having simplicity as a major underlying goal. Under the UNIX system, for example, the concept of security is reduced to a very basic level. The entire kernel runs at one privilege level and user process are allowed to access the kernel routines by making system calls, and (for this purpose) all user processes are given equal access. On the **MULTICS**[cej] operating system (developed chiefly by the Massachussets Institute of Technology and Honeywell Information Systems Inc.) there are four levels of privilege

for the operating system and, below these, four privilege levels for user processes. This system achieves a high degree of security but it does this at the expense of having to administer many large tables of privilege information. In an operating system which is required to give a fast response times which this project requires, freedom from such lengthy manipulations would be an advantage.

In addition to the technical features discussed above there is one other non-technical feature of the UNIX system which also makes it a good choice as the target operating system for this project and that is its popularity. Since its release, the UNIX operating system has been implemented on a large number of highly different machines and has given rise to a number of 'lookalikes' (for example IDRIS and XENIX) which have found ready markets. One of the aims of this project is to provide a more amenable interface to real-time control systems and it would therefore be an added bonus if this interface is one which is already familiar to a large number of people.

4.1.2 The Disadvantages of the UNIX System

Although the UNIX operating system is, in many ways, well suited to the demands of this project, there are a few deficiencies which make it less than ideal. As stated above, a wide range of I/O devices can be interfaced to the UNIX kernel. Devices are considered to be either block devices (like a disc drive) or character devices (like a teletype terminal). There is, however, no concept in the UNIX system of a device being another processor. Though it may not be difficult to modify the UNIX kernel so that it recognises a second processor as a unique piece of equipment, one of the strongest recommendations of UNIX is that it has *no* unique pieces of equipment of which the user is aware (a memory

management unit is unique, but totally transparent to the user) and provides the same user interface for all devices.

In order to convert the UNIX operating system to a multi-processor system, therefore, it would be necessary to write an extensive driver which would allow a processor to be seen by the UNIX kernel as a block or character device.

Another feature of the UNIX system is that user processes running under it have only a limited conception of real time. Utilities are provided to allow a process to calculate the amount of time it has spent in any section of code, but there is no way that a process can ensure that a given section of code will execute in less than some critical time limit. This is due to two operating system activities: (a) the process may be preempted or (b) the process may be swapped to disc.

A process running under an operating system like International Business Machine's OS/MVT[23] (before the Timesharing Option was added) could be certain of running continuously to completion once started unless it made an error, made an I/O request or another process with a higher priority was queued. If an error-free process was therefore given the highest possible priority, and made no I/O requests during its critical sections, it would have been possible to accurately predict the amount of time that would be spent in the critical sections.

Under the UNIX system any user process may be preempted if it has run for a period of time and there are other processes queued. This occurs regardless of the original priority of the process as the UNIX kernel dynamically adjusts the effective priority of all processes, favouring those which have been queued longest without being run.

The problem of a process being swapped to disc is somewhat different as it depends not on the priority of the process, but on the

memory requirements of all the current processes (including its own). The VAX/VMS operating system (developed by the Digital Equipment Corporation) allows processes to be given priorities greater than the priority of the swapping process but the UNIX kernel grants the swapping process (which runs at the kernel privilege level) the highest possible priority in the system. Nevertheless, a process's core image may be locked against swapping, but this can be done only from within the kernel or (on some versions of the UNIX system) by a special system call which the kernel may choose to ignore as it could lead to the system becoming deadlocked.

4.1.3 The Development of the UNIX Operating System

The original UNIX system was written in 1969-1970 at AT&T Bell Laboratories to provide a flexible and powerful environment for software development. It was intended to run on a PDP-7 computer and was written in assembly language. Later, a version was written for the PDP 11/20 computer. All of the systems were single-user and none was ever released outside of AT&T.

By 1971, the system had been completely rewritten in the C programming language and was running on all machines in the PDP range from the PDP-11/34 to the PDP-11/70. This was essentially the first version that looked like the UNIX system of today and was known by the version of the manual which accompanied it - Version 6.

Due to the anti-trust laws in the U.S.A., AT&T were not able to profit from the UNIX system and consequently little effort was put into providing a fully-supported version. Nevertheless, AT&T offered academic institutions the complete source code for the UNIX system and its utilities for a nominal fee. This policy gave rise to the Berkeley 4.1

and 4.2 versions of the UNIX system which were started at the University of California in 1975.

In 1979, the most widespread version of the UNIX system was released. This was Version 7 and it has provided the basis of many of the 'lookalike' systems today.

Having overcome the legal difficulties, AT&T released the System III UNIX system in 1981 as their first commercial, supported version. In 1983, this was superseded by System V and the current release at the time of writing is System V.3. In the intervening years, many variations of the "standard" UNIX system have been produced (e.g. Programmer's Workbench UNIX[1,2,3]). At present, AT&T are attempting to define System V as being the one standard, and versions of the system by other companies must pass an AT&T validation test before they can be released.

The project on which this thesis is based began with the Version 7 system, but soon moved to the System V system. In the descriptions which follow, contrasts are made between the two systems. It should be noted, however, that some features which appear in the description of the System V system were actually included on some of the earlier versions of UNIX systems which were not used in the project.

4.2 A General Description of the UNIX Version 7 Operating System

4.2.1. Exception Handling

When an exception occurs, control is passed to a kernel routine written in assembly language along with a pointer to an exception handler (usually written in C). The kernel routine saves the state of the current process and transfers control to the handler. Upon a successful return from the handler, the state of the current process is restored and its execution resumed if the exception occured while the process was

executing in the kernel (e.g. during a system call). If this was not the case, the process can be preempted if a scheduling flag is set.

Unresolveable exceptions (e.g. divide by zero) while a process is not in the kernel cause a signal (see below) to be sent to that process which may terminate it. If such exceptions occur in the kernel, all outstanding data is written to the appropriate files and no further activities are allowed.

4.2.2 Process Management

A process in the UNIX system consists of four sections:

(1) The process header, which contains all the information about the process which is not needed when the process is swapped out to disc (e.g. pointers to the arguments of the current system call or the values of stored CPU registers). Such variables only take up a small proportion of the space allocated for the header and the remainder of the space is used as the kernel stack area, thus effectively providing the kernel with a separate stack for each process. The process header is not directly addressable by the user process.

(2) The text segment, which contains the re-entrant, executable code for the process. This segment is not allowed to grow while the process is running. If the code is not shared between processes then this segment is combined with the data segment.

(3) The data segment, which contains the read/write data locations required by the process. Both the initialised and the uninitialised data is contained in this segment. This segment begins at the first convenient address ('click' boundary) above the text segment and can grow towards higher addresses by using the ***break()*** system call.

(4) The stack segment, which contains the stack of the process when it is running in the user state of privilege. This segment begins at the highest possible (virtual) address and is able to grow towards lower addresses if an overrun occurs. It depends on the hardware whether this growth occurs automatically or whether a system call must be used (in which case the overrun must be anticipated).

Information about the process which is needed even when the process is not resident in primary memory (e.g. the event which will cause it to resume execution, or its address in the swap area) is held in a table in the kernel data area. If the process contains a shared text segment, information about this segment is held in a separate table in the kernel data area. Entries in the process header, process table and text table are all cross-referenced.

At the lowest level, processes are handled using two system primitives *save:* and *resume:*. When a process suspends execution all volatile information about its state (e.g. the processor's internal registers) must be saved until the process is ready to resume execution. The kernel calls *save:* which stores all this information in a location in the process header along with the return address of the subroutine call. When *save:* returns, the kernel calls *resume:* if another process is ready to execute otherwise it calls the *wait:* primitive which does nothing until an interrupt occurs.

When *resume:* is called it fetches the volatile information from the location in which *save:* stored them, and marks the process header as being that of the 'current' process and locates the kernel stack pointer to the top of the stack for this process. It then returns as if the call to *save:* had returned but with a different value.

4.2.3 Memory Management

The UNIX kernel is permanently resident in memory and occupies the lowest memory locations. All memory above the top data location of the kernel is available for user processes. As stated above, the kernel stack is placed in the process header of the current process. All user processes may be swapped out so there is also a process header for the swapping process in which the kernel stack is placed when there are no other processes running.

Initial memory requirements for the text and data areas of the process are held at the beginning of the file from which the process is loaded. An estimate of the size of the stack is made from the amount of data which will be placed on the stack (the 'environment') before the process is started.

The kernel maintains a table of free memory in which the base address and size of free areas of memory are held in order of increasing base address. Memory is allocated in the UNIX system in multiples of a fixed number of bytes called a click (which may be between 64 and 512 bytes depending on the system) using a first-fit algorithm. When an area of memory is freed, its description in the table is merged with those of any free areas on either side.

If a process has to increase the size of its data or stack segments, that segment must be copied to a new area of memory. (If a segment decreases in size, the free portion is simply deallocated). If no free area of memory exists which is large enough to contain the segment, the process is swapped out. The swapping process is run whenever an area of memory is freed, so the process will be swapped back into memory when a suitable space becomes available.

Processes can only be created by making a copy of an existing process (forking) so, if there is no available memory when a new process is being made, the old process is swapped out to disk without removing the core image, thus creating two copies. Again, the copy on disk will swapped in when memory is made available.

If a pure (shared) text segment is allocated space in memory, a copy of the segment is also placed in the swap area. The text segment can therefore be "swapped out" simply by freeing its area of memory, which increases the efficiency of the system. The memory is freed only if all processes which share the text segment are currently swapped out. The segment image on the swap device is only freed when all processes sharing it have terminated.

If no processes are swapped out to disk, the swapping process is effectively dormant. When one or more processes are swapped out, however, the swapping processes repeatedly tries to swap in the largest. If there is no available memory it searches the process table for the largest process in core which is awaiting some event with a low priority and swaps that process out if it finds one. If there are no such processes, it finds the process which has been in memory the longest and swaps it out if it has been in memory longer than the target process has been out of memory.

The available space on the swap device is managed in exactly the same way as core memory. If there is no suitable slot available on the swap device when the kernel which is to swap a process out, the system is deadlocked and behaves in the same manner as when it handles an unresolveablve exception in the kernel.

The Version 7 UNIX kernel does not support virtual memory, therefore very large processes must employ some sort of overlay mechanism (see

below) as the kernel imposes an upper bound on the size of processes and the *break()* call fails if processes attempt to exceed this limit. Similarly, if the attempted growth is due to an *exec()* call (see below), that call would fail.

4.2.4 Processor Management

As mentioned above, the only manner in which the UNIX operating system creates new processes is by forking. The core image of the new process is therefore an exact image of the parent process except for a few variables in the process header which must reference a new entry in the process table.

If a new core image is required, the *exec()* system call is used, in which case the entire core image is overlaid with that of the new process. Thus, if a process uses the *exec()* system call without first using the *fork()* system call, the entire core image of the parent process is lost and cannot be referenced again. (The exception to this is when the new process is marked as an '*overlay*' process, in which case only the text segment is overlaid).

Processes are destroyed either by using the *exit()* system call or upon receipt of a signal (from the kernel, another process, or itself) for which it has no handler. The parent of a process which terminates is sent an indication of the child's termination status. If the true parent terminates before the child, the child process is regarded as being a child of a special process called '*init*' which is discussed below. A process which terminates is not immediately removed from the process table but remains until the scheduler has completed the necessary accounting updates and its parent (or *init*) has acknowledged its termination.

When a process is ready to execute it is placed in a queue of running processes. This queue is examined approximately once per second and the process on the queue which is in memory with the highest priority is made the current process and is allowed to execute. This process continues to execute until it has to await an event (e.g. the completion of a I/O transaction) or it terminates or the queue is examined again (approximately one second later) and it is no longer the process with the highest priority.

The base value of a process's priority is set by the kernel and depends on the process's current activity. For example, if a process is currently in the middle of being swapped out to disk, it is given a very high priority as this activity must be performed as quickly as possible to prevent the system becoming deadlocked. If, however, the process is performing no special activity (it may be performing some lengthy calculation which requires no access to kernel facilties) it is given a relatively low priority.

If the priority of a process is above a certain minimum priority (i.e. it is performing some special activity) its priority remains fixed until that activity is completed. In fact, if its priority is very high indeed, the process cannot be disturbed at all (e.g. by sending it a signal) except by an event signifying the end of that activity.

Low priorities, however, are dynamically adjusted by the scheduler if the process is waiting in the queue to be executed. When a process's priority is low its effective value depends not only on its current activity, but also on its '*niceness*'. This is a value that can be set by the process (or inherited from its parent process) in order to increase or decrease the proportion of processor time which it will be given in relation to the other processes in the system. (The kernel may disallow

very low 'niceness' values if the owner of the process is not the 'super-user'.

Due to the increase in the priority of each waiting process every time the scheduler examines the queue, each process is guaranteed some processor time as it will eventually be the first process in the queue whose priority is greater than or equal to the priorities of all the other processes in the queue. If there are a large number of high-priority processes and relatively few low-priority processes, however, the the low-priority processes will be scheduled very infrequently.

4.2.5 Device Management

Some components of the computer system's hardware are made completely transparent to the user by the UNIX kernel. The user has no direct contact with the allocation of the processor or the available memory, for example. Other units of hardware are presented to the user through a uniform interface. The UNIX kernel manages these devices by dividing them into two classes - block devices and character devices - and holding two tables in memory of the routines which need to be called for each device for which I/O is requested by the user.

Block devices are those where data accesses are inherently structured towards large numbers of bytes per transfer - disks, drums or magnetic tape - whereas character devices are, in theory, those where data accesses are performed on single or few bytes per transfer. In practice, however, character devices tend to any device which uses unstructured I/O transfers and even disks may be referenced as character devices (in order to reduce the number of accesses required when tranferring a large stream of contiguous data, for example). It is not

unusual, therefore, for drivers of block-structured devices to contain routines for both block and character operations.

The basic routines by which a process accesses a file or a device (which are treated as files) under the UNIX system are *open()*, *close()*, *read()*, *write()* and *seek()*. Of these, *seek()*, which allows the process to access a file in a non-sequential manner, requires no hardware support as no I/O is performed and all that is involved is an adjustment in the value of a pointer variable. It should be noted, however, that some devices (e.g. a magnetic tape drive) are incapable of seeking and despite the uniform interface, this operation would be meaningless if used with such devices. The remaining four routines are implemented by calling a device-dependent routine to implement each system routine.

On some systems, a programme '*mkconf*' is provided to assist in generating the exception vectors and device tables.

4.2.5.1 Block devices

For block devices, the kernel consults a table which contains, for each device, pointers to the following device-dependent routines: *dev_open()*, *dev_close()* and *dev_strategy()*. Each device is identified by a major and a minor device number and the kernel uses the major device number to index the device table while the minor device number is passed to the device-dependent routine for its own interpretation. If any of the device routines are unsuitable for the particular device (e.g. a device may be read-only) they may be replaced with one of the two dummy routines *nullsys()* and *nosys()*. *Nullsys()* does nothing and returns successfully while *nosys()* does nothing and returns an error code.

When a process wishes to access a block device the kernel calls the appropriate *dev_open()* by indexing the device table with the major device

number. After the device has been successfully opened (and the interpretation of this is device-dependent), I/O transactions may be performed. All I/O transactions to and from block devices are performed in units of 512 byte blocks and the kernel interface is via the appropriate *dev_strategy()* routine. Normally, this routine merely checks that the transfer request is valid and places the request in a queue. On return from this routine the process suspends its execution until it is informed that the transaction is completed.

The block device driver must contain other routines to actually cause the device to perform the transaction when it reaches the head of the queue and to take the appropriate action when the low-level exception handler processes an interrupt from the device. Each device driver manages its queue of requests in its own way but, as the requests are not scheduled by the system the only opportunities for queue manipulation are when the routines are explicitly called by the system to initiate a transaction or when the device interrupts.

The device-dependent *dev_close()* routine is only called when all processes which open the device have either terminated or have executed the system call *close()* for the device. As for *open()*, the interpretation of what constitutes a closed device is left to the individual driver.

As all requests for block device I/O are divided by the UNIX kernel into 512 byte requests, if a process wishes to transfer a large block of data to or from the device, this will be translated by the system into a large number of 512 byte requests. If the data resides or is to reside in a contiguous area on the device, many devices would be able to process the request more quickly by treating it as one or a few large requests. For this reason it is often convenient to access a block-structured

device as a character device in order to minimise the time needed for large data transfers.

4.2.5.2 Character Devices

The UNIX kernel maintains a separate table for character devices which is indexed in the same way as the block device table. The relevant device-dependent routines for character devices *dev_open()*, *dev_close()*, *dev_read()*, *dev_write()* and *dev_ioctl()*. The *dev_open()* and *dev_close()* routines are used in the same way as for block devices while *dev_ioctl()* is called upon execution of the *ioctl()* system call. This system call is intended to enable the process to exercise some degree of control over the operation of the device (e.g. to set the rate at which characters are processed by a serial interface). The effect of this call depends upon the scope and flexibility of the *dev_ioctl()* routine and this particular part of the device interface cannot be described as uniform.

The *dev_read()* and *dev_write()* operations perform the actual I/O transactions to and from the device according to a byte count and a pointer to the start of the data which are provided by the system. Again, each device driver is free to interpret these values in any way that is suitable for the device. It is not uncommon, for example, for the read and write routines of devices that are, in reality, block-structured, to call the *dev_strategy()* routine for the block device to handle the transaction. If the transaction is not "instantaneous" then some method of queuing must be employed and an interrupt routine provided.

4.2.6 Information Management

A file in the UNIX system is a one-dimensional array of bytes. The UNIX operating system maintains all its files in a hierarchical structure

of directories and files. The distinction between the two is not very great as directories are simply files which can only be written to by the system. The top level node in the hierarchy is a directory called the *root directory* and resides on a block device known to the kernel as the *root device*. From this node, it is possible to reference every other node in the filing system.

The root device (and all other mountable devices) can be divided four sections, each consisting of a number of 512 byte blocks. The first block on the device (with logical block number 0) is the *boot block* and is intended for the system bootstrap programme. The second block on the device is known as the *superblock* and contains all the information about the block structure of the device (e.g. the total number of blocks on the filing system or a list of free blocks).

The third section of the device typically occupies about a quarter of the total space of the filing system and consists of groups of data called '*i-nodes*', each of which refers to one file and contains all the information regarding the structure of that file on the device. The remainder of the storage area is free for allocation to files.

While the superblock contains all the information about the block structure of the device, it is the *i-list* (the list of i-nodes) which allows files to be referenced in a way which is independent of the physical structure of the filing system. Each block in the i-list contains eight i-nodes. The first i-node in the i-list is unused, while the second is that of the root directory of the filing system. All other i-nodes can be allocated and freed as necessary. The i-node contains the following information about the file:

(a) The times of its creation, last access and last modification.

(b) The type of file: a file can be a directory, a block or character device, a regular file or a pipe. If the file is a device the i-node also contains the major and minor device numbers.

(c) The user and the group that the file belongs to.

(d) The access permissions of the file: which users and which groups are allowed to read, write, or execute the file and (if the file is allowed to be executed) which user or group permissions the resulting process will appear to have.

(e) The size of the file in bytes.

(f) The logical block numbers on which the file is stored. These block numbers are stored in an array of 13 block numbers, the first 10 directly reference the first 10 blocks of the file. If the file is larger than 10 blocks (5120 bytes), the block referenced by the eleventh block number is an *indirect* block which contains the block numbers of a further 128 direct blocks. If the file is even larger than 138 blocks (70,656 bytes), the twelfth block number references a *double-indirect* block which contains the block numbers of 128 indirect blocks and if the file larger still (than 8,459,264 bytes) the last block number references a *triple-indirect* block which contains the addresses of 128 double-indirect blocks. The UNIX system therefore caters for files up to a maximum of 1,082,201,087 bytes in size, nevertheless, if the file is small it can be quickly accessed as most of it will be referenced by direct block numbers.

All multiple file systems contain the same structure as that of the root file system. Each has an i-node for the root directory and the parent directory of the root directory for the file system. If the file system is unmounted, or if it is the root file system, the parent

directory and the root directory point to the same place on the disk. Any file system may be mounted, however, "under" a directory of another file system. In this case the parent directory becomes the directory under which the file system is mounted and all files on the mounted file system can be referenced from the root directory of the root file system as if all the files resided on one file system and not on two different devices.

The structure of a UNIX directory is very simple containing entries consisting of the file name (14 characters maximum) and the number of the i-node. The root directory is referred to as '**/**' and any files under this directory are referred to as '**/file_name**'. This nomenclature is recursive so that a complete file reference in the UNIX system has the structure '**/dir1/dir2.../dirx/file_name**' where the '*dir<n>*' are directories. Such a structure is termed a "pathname" as a complete path is traced from the root node to the referenced node.

Files on a UNIX file system may be shared by the process of *linking* (though only the super-user may link to a directory). When a link is made to an existing file, a new directory entry is made but it points to the i-node of the existing file. The link-count in the i-node indicates how many directory entries use that i-node. Files are created using the ***creat()*** or the ***mknod()*** system call (to create regular and special files, respectively), and deleted using the ***unlink()*** system call. This call removes the directory entry and decrements the link-count. When the link-count becomes zero the i-node is freed.

Files are referenced by user processes by their pathname, and are accessed using a '*file descriptor*', which is a number, returned from a successful call to ***open()***, ***pipe()*** or ***creat()***, which is used by the kernel

to index a table of pointers to i-nodes which is maintained in the process header.

4.2.7 Bootstrapping

The UNIX Version 7 operating system uses a multi-stage bootstrapping process to load the kernel into memory and start its execution. The first level of the procedure may reside either in the boot block of the root device or in the read-only memory (ROM). In the former case, some extra provision must be made to read this block, load the contents into memory and start its execution.

The first stage bootstrap is written in assembly code and contains procedures to communicate with the system console (usually a serial port) and the device from which the second stage of the bootstrap is to be loaded. It executes at a location in memory at which there is no danger of its being overwritten when the second stage is loaded. The first stage must initialise these two devices and prompt the operator for the pathname of the second stage. The pathname can be arbitrarily long, but no file in the path may have an effective length of more than 138 blocks as the search algorithm can only search one indirect block. For example, if the file being examined is a directory whose actual length exceeds 138 blocks, but in which the entry being searched for appears before this limit is reached, the algorithm succeeds. When the target file is found, and is seen to be marked as executable, it is loaded into memory and control passes to the start of it.

The second level bootstrap is written mostly in C and contains a complete path searching algorithm. In addition it knows about all the I/O devices in the system and must initialise any which need to be initialised as there is no provision in the kernel for this. The second

level prompts the operator for a pathname which included the name of the device whose filing system is to be searched. It loads the file (containing the kernel) into memory and passes control to it.

When the kernel begins executing it positions the stack pointer to a safe place, clears all memory locations which it is not using and begins process initialisation. Initialisation includes the clock, the memory management unit and the system tables. No processes are running at this stage so the first slot in the process table is initialised with the variables for the swapping process and memory is allocated for its process header. The code for this process resides in the kernel and when entered it immediately forks, making two processes. The parent (swapping) process never exits but when the child is executed enough memory is allocated to load a sequence of machine language instructions which perform the *exec()* system call. After the *exec()* is performed creation, termination and scheduling of processes proceeds as described above.

The process which is loaded upon execution of the first *exec()* is called '*init*'. When first executed it opens the system console device for reading and writing and forks to create a 'shell' for this terminal. *Init* does nothing until this process terminates. The shell allows the creation of processes from the console, but as this is the only active terminal in the system, the system is said to be running in single-user mode.

When the single-user shell terminates *init* is informed and enters multi-user mode. It forks to create a '*getty*' process for each terminal device in the system which is marked as 'ON'. If any of these child processes terminate *init* forks to restart it. Each terminal of the

system is, therefore, always connected to the operating system via a process.

Getty sets the characteristics for the terminal from a table and *execs* to start the process '*login*'. *Login* prompts for a user name and password and, if these are correct, *execs* to provide a shell for the terminal. As mentioned above, when the shell terminates, *getty* is restarted by *init* and the cycle is repeated.

The *init* process remains in multi-user mode until it receives the '*hangup*' signal from another process. When this occurs, it terminates all its descendant processes and re-enters single-user mode. Multi-user mode can be resumed by the process described above.

4.2.8 UNIX System Calls

Throughout this chapter, mention has been made of system calls. By using UNIX system calls, user processes are able to access the kernel facilities and the structure of the system call ensures that such accesses are performed in a controlled manner.

When a process wishes to execute a system call, it stores the arguments for the call in its stack segment and stores the numerical identifier of the specific system call which it wishes to access in a known register. It then initiates a system call exception. The kernel is entered due to this exception (see above) and the kernel's generic exception handler calls the appropriate kernel routine. Upon completion, control is returned to the process. The system returns an error code to the calling process which indicates to the process whether the call was successful or not. If the call failed the error code gives some indication of the reason for the failure.

There are about 30 system calls in the UNIX system. They may be grouped in a few general categories:

Process Information: These calls return various process identifiers as its process identification number ('*pid*') or the user and group which own it or initiated it.

Process Control: These calls attempt to control the execution of the process by changing the size of its core image, creating new processes (*fork()* and *exec()*), terminating the process, handling signals, adjusting its priority or preventing it from being swapped or enabling its execution to be traced by its parent.

File Information: Such calls enable the user to determine the type and accessibility of files.

File Control: These calls allow the user to manage all the accessible files. Files may be opened, closed, read from, written to, created, deleted or linked to other files. In addition the owner and type of the file may be set, file systems may be mounted and unmounted, and the operation of devices controlled.

Process Communication: These calls allow processes to send signals to other processes and to exchange data with them.

System Control: These system calls read system variables such as the time and cause the system to perform internal operations such as writing out buffered data or setting its idea of the time.

4.2.9 Inter-Process Communication and Synchronisation

Communication between processes is of vital importance in a multi-tasking operating system. While it is possible for each process to be run in isolation, considerable savings are made (both in the ease of programme development and in efficiency of execution) when co-operating processes are executed in parallel and share information. To achieve efficient and unambiguous communication receiving processes must know when data is available for them, and sending processes must know when the receiving processes are able to accept data. The concept of communication is therefore inherently bound to that of synchronisation.

Communication in the UNIX Version 7 operating system is achieved by two methods: *signalling* and *pipes*. (A third method - *multiplexed files* - has been experimentally introduced on some implementations but, as this is not yet a standard feature it is not considered here). The system calls which are used to achieve communication are ***signal()***, ***kill()***, ***alarm()***, ***pause()*** and ***wait()*** (for signals) and ***pipe()*** along with the normal file-control calls (for pipes). No data is exchanged when a signal is sent, so signalling is normally used only as a synchronisation measure.

4.2.9.1 Signals

Signals can be sent from the kernel to any process (or group of processes) or from any process to another (using the ***kill()*** system call), provided they are owned by the same user or the sending process is owned by the super-user. The kernel signals the process on detection of an abnormal exception (e.g. an illegal instruction or a segmentation violation) In total the UNIX Version 7 operating system allows 16

separate signals to be sent of which 15 have standard, pre-defined meanings.

Whenever a process enters the kernel (either due to a system call or when its state is changed by the scheduler) it is informed of any signals pending for it. Normally, when a process receives a signal, the signal causes the process to terminate. A process can, however, choose to ignore the signal, in which no action is taken, or it can specify a user-provided routine to handle the exception. In this case the signal acts as an interrupt to the process, on receipt of which it can take whatever action it likes. (There is one defined signal which can never be caught or ignored and this provides a sure way of achieving process termination).

Signals provide a way for processes to handle abnormal conditions intelligently, without termination. By using the *pause()* and *wait()* calls, however, a process can synchronise its execution with other events. *Pause()* suspends execution indefinitely until a signal is received while *wait()* acts like *pause()* except that execution continues if a child process terminates. *Alarm()* signals the calling process after a specified time interval has elapsed.

4.2.9.2 Pipes

A UNIX pipe is essentially a file with a fairly small maximum length (typically about 2kb) which allows data to be written to it and read from it in a first-in, first-out manner. As with all files on the UNIX system, the user is unaware of the detail of the implementation of this communications channel and communicates with it using the normal *read()* and *write()* operations. (Seeking is obviously prohibited). A pipe is established using the system call *pipe()* which acts somewhat like

creat() and *open()*, except that no file name is specified and two file descriptors are returned. One of the descriptors refers to a read-only file, while the other refers to a write-only file, but both descriptors reference the same i-node.

Synchronisation between reading and writing processes is maintained by the kernel and is transparent to the process. When the pipe is full, any process which attempts to write to it has its execution suspended until another process removes some of the data. When the pipe is empty, processes which attempt to read from it are suspended until data is written to the pipe by some other process.

4.2.10 Utilities

The UNIX operating system is supplied with an extensive range of utility programmes, which handle sorting, text editing, word processing, string searching, character translation, equation and table formatting, as well as programmes which directly interface to the kernel system calls to provide functions such as copying and renaming files, linking to files, etc. Using the pipe mechanism many of these programmes can be executed in parallel to easily implement 'customised' commands. There are also a number of special 'system commands' for performing activities such as checking file systems, checking the system accounting and controlling the operation of system daemons.

In addition to these 'system commands', a large library of routines is provided which are intended to simplify the incorporation of generally useful features (e.g. string handling) into user-written programmes.

One system command which is particularly useful is the standard shell. This is essentially a CLI which interprets all lines of input as the name of a file to be executed and a list of arguments to be passed to

the resultant process. The shell forks and calls *exec()* to achieve this. Certain commands have to be executed directly by the shell without forking (e.g. the command which changes the user's current 'working directory').

The shell provides a number of features to facilitate the use of various kernel functions. For example, it provides a simple syntax for the creation of pipes and attaching processes to them. Processes which are to be run in parallel with the shell can be simply created by instructing the shell to run them in 'background mode'. In this case, the shell calls *fork()* and *exec()*, but does not call *wait()*.

File name generation is made easy by the expansion of several 'meta-characters' so that processes which would take a long string of filenames as arguments can be easily initiated. It is not always necessary to specify the complete pathname of an executable file to the shell. As most commands tend to be grouped in a few directories, the shell can be instructed to search a list of directories in a specified order until it finds a matching file name. A complete pathname is only needed when the file resides in a directory which is not in the list or when more than one file of the same name exists in directories on the list and it is required to force the execution of a particular one.

Shell commands may be grouped together to form more complicated commands and the shell provides a language syntax in which groups of commands may be structured using **'if-then-else'** blocks and **'for'** loops. Variables which are considered to be useful to all processes started by the shell (global variables) can be defined and the shell places such variables in the environment (stack) of each process using *exec()*.

4.2.11 Users

Information on users of a UNIX system is kept in a 'password' file. Each user has a user name and a user identification number (id.). The password file contains, along with this information, an encrypted version of the user's password, the name of the user's 'home' directory, and the pathname of the file which *login* will *exec* when the user logs into the system i.e. the file to use as the shell.

Each user is also assigned to a group, the id. of which is also held in the password file. Information about the group is held in another file containing the group name, the group id., the group password (encrypted) and a list of users who are allowed to use the group.

Files created by the user are owned by him. The files are also marked with the owner's group id. and the accessibility of all files is determined on the basis of these two identifiers.

Processes normally acquire the same access permissions as the user who initiated them. Executable files can, however, be marked by the owner so that, upon execution, the process always acquires the owner's access permissions and/or those of his group. This feature permits users to allow access to their files in a controlled manner and is used, for example, by *login*. *Login* writes to various system record files owned by the super-user (e.g. '/etc/utmp', the record of users currently logged in). Normally, access to these files is denied to other users. If the user runs *login*, however, the process acquires the access permissions of the super-user and is therefore allowed to write the correct information into the files.

4.3 A General Description of the UNIX System V Operating System

The UNIX System V operating system is a development and expansion of the older UNIX Version 7 operating system described above. Many of the features of the Version 7 system have been retained in the System V system, therefore, only those areas of System V which are different to those of Version 7 will be discussed in detail.

4.3.1 Exception Handling

Exceptions are handled by the UNIX System V kernel in the same way as in the UNIX Version 7 kernel.

4.3.2 Process Management

Processes running under System V may have a fifth segment in addition to those described for the UNIX Version 7 system. This is a data segment which may be shared between processes and is known as the '*shared memory*' segment.

4.3.3 Memory Management

The memory management of System V is essentially the same as that of Version 7, although the provision of shared memory segments necessitates some changes in various kernel routines. An attempt has also been made to make the system less susceptible to deadlock due to the exhaustion of swap space by making it possible to free areas of the swap space which are occupied with pure text segment images (see above).

When swap space is low, the kernel searches the table of text segments and frees the swap space of those segments which still have a core image. A flag is then set in the table to indicate that, should it be required to swap the text segment out, a true swapping operation must

be performed. In this way, a compromise is made between speed of operation and storage requirements.

4.3.4 Processor Management

Processes are scheduled under System V in the same way as for Version 7 except that two swapping processes run in the kernel - one concerned only with swapping pure text segments. Queue hashing has made process searches faster for processes awaiting events.

4.3.5 Device Management

The basic concepts of the UNIX device system have remained unchanged. Both block and character devices are now able to have an initialisation handler and a '*powerfail*' handler. The configuration programme '*config*' is more powerful than the UNIX Version 7 '*mkconf*' and handles the complete generation of the low-level device interface.

4.3.6 Information Management

The structure of the System V filing system remains unchanged although the logical block size has been increased to increase the speed of block tranfers. Older file systems are still supported though with decreased efficiency. A variable in the superblock holds the size of the filing system.

Due to the increase of block size, larger files can be supported as the indirect blocks of each i-node can now each point to a larger number of blocks.

4.3.7 Bootstrapping

Due to the increased block size, no intermediate-level bootstrap programme is needed as one indirect block is sufficient to address the whole of the kernel. No unnecessary initialisation need be built into the bootstrap programme as the kernel now supports device initialisation.

As mentioned above, two kernel swapping processes are initiated. ***Init*** is now able to assume seven run-levels (one single-user and six multi-user) rather than just two. The processes which ***init*** creates at each level are controlled by a table which indicates when the processes are to be created (e.g. initially or upon receiving a powerfail signal) and whether they are to be restarted upon termination.

The current run-level of ***init*** is changed by running the process with the run-level as argument.

4.3.8 System Calls

The available system calls have been augmented by calls to exploit new features of the system (e.g. shared memory segments) and some older calls have been given a more flexible structure. Using ***open()***, for example, the user can specify that the required file be created if it does not already exist. Under the older system, the ***open()*** call failed in this case, and a call to ***creat()*** had to be made.

4.3.9 Inter-Process Communications

The major advances of System V over Version 7 is in the area of inter-process communications. In addition to the above-mentioned facilities of signals and pipes, System V now also supports *named pipes*, *shared memory*, *inter-process messages* and *semaphores*. The number of

signals has been increased to nineteen where certain of them are user-definable.

Pipes may be created as before, but it is now possible to name the file which is used as a pipe, thus increasing its accessibility. Processes using the pipe after its creation may attach to it using the normal *open()* system call.

Shared memory segments, message queues and semaphores are all distinguished by an inter-process communication (IPC) key. These keys are determined by the process which creates the communications channel. Processes may then attach themselves to that channel by passing the key to the kernel. If the operation succeeds, the kernel returns an IPC identifier to the process in the case of message queues and semaphores. This is used in the same way as a file descriptor and is used, to reference the channel. In the case of shared memory segments, an address is returned by the kernel and this segment is then accessed in the same way as the data segment of the process.

Care must be taken that processes do not attach themselves to a spurious communications channel by using a key which is already in use by another group of processes. To this end, a utility '***makekey***' is provided which attempts to provide a unique key for each group of processes. If all processes using the channel stem from the same parent, a private key may be specified. Using this method, each process inherits the IPC identifier when the parent forks and the channel is guaranteed to be free from extraneous activity.

Shared memory segments allow a group of co-operating processes (termed a '*project*') to access a common area of memory and hence share information. The segment is mapped into the data area of each process which is linked to it and is accessed in the same manner as the normal

data segment. It is therefore the responsibility of the project to ensure mutual exclusion of accesses to the same location in the shared memory segment.

Message queues are maintained by the kernel and each process in the project may send or receive messages via the queue. Each process may decide to suspend execution until a message arrives, or to continue immediately if no message is in the queue when it makes a request to the kernel.

The semaphore mechanism implements Dijkstra's algorithm[11], and each project may have multiple semaphores.

4.3.10 Utilities

The standard UNIX Version 7 utilities are incorporated into System V along with a few new ones. A full screen editor has now been made a standard utility, and a more powerful filesystem checking utility is included.

4.3.11 Users

Users of the UNIX System V operating system are managed in the same way as users of the UNIX Version 7 system.

Chapter 5

The Implementation of the UNIX Operating System

5.1 The C Programming Language

The bulk of the code which comprises the UNIX operating system is written in the C programming language[22]. Sections of code which require optimal execution times (e.g copying of large blocks of data) and sections which have no equivalent in C (e.g. writing to a specific internal register of the processor) are written in machine code, but the number and sizes of such sections are kept as small as possible to provide maximum portability of the operating system.

The C programming language was developed in the early 1970's and was greatly influenced by the typeless BCPL programming language[36] which was developed at the Massachusetts Institute of Technology and Cambridge University. An interpreted version of BCPL called "B" was used in the initial development of the UNIX system and this was then expanded to include types and a machine code generator for the PDP-11 computer and became the language "C". C compilers are now available for a large number of machines and operating systems.

A C program consists of keywords, identifiers, operators and punctuation. There are some 30 keywords and these are not available for use as identifiers. Identifiers are sequences of letters and digits starting with a letter. The character '_' is considered to be a letter and upper case and lower case letters are distinct. Identifiers are used for variables, labels and function names and are delimited by spaces, tabs, newlines, operators and punctuation.

The syntax and facilities of C have varied somewhat since it was first developed. In the description below is of the version of C which is supplied with the UNIX System V operating system.

5.1.1 Types and Constants

C provides a number of simple data types which determine how the object in the storage location referenced by the identifier is interpreted. The type ***char*** is the smallest type. All other types are stored in sizes which are multiples of the size of ***char***. When a constant refers to an actual character, it may be referred to as that character enclosed in single quotes (e.g. 'c'), by an escaped character in single quotes (e.g. '\n' refers to the newline character), or by an escaped (octal) three-digit sequence in single quotes (e.g. '\015' refers to the ASCII carriage return character). (Character specification by the last method is dependent on the character set being used). A number of characters enclosed by double quotes is taken to be a constant string and is stored as the indicated characters plus the '\000' character which indicates the end of the string.

Integer values are available in three sizes: ***short int***, ***int***, and ***long int***, although not all of these may actually give a different storage size on some machines. Integer constants may be expressed in decimal, octal or in hexadecimal. All integer constants are a sequence of digits. Decimal constants have no prefix, octal constants are prefixed by '0' and hexadecimal constants are prefixed by '0x'. Integer values may be specified as being unsigned. On some implementations, ***char*** values may also be unsigned.

Single precision floating point numbers have type ***float*** while double precision numbers have type ***double***. Floating point constants contain

either a decimal point or an exponent part (prefixed by 'e' or 'E') or both.

C also allows the specification of enumerated types along with the possibility of defining the integer value of each member of the type. There is, however, no provision for automatically iterating over all the members of the type.

5.1.2 Expressions and Operators

Expressions in C are formed by combining variables, references to variables, constants and function calls using binary or unary operators. Most binary operators are left associative. Exceptions to this will be noted in the following description.

The arithmetic operators are + (addition), - (subtraction), * (multiplication), / (division) and % (modulus). When positive integers are divided, the result is truncated towards 0 whereas if either of the operands is negative, the result is machine dependent.

Comparison is achieved via the operators > (greater than), >= (greater than or equal to), < (less than), <= (less than or equal to), == (equality) and != (inequality). Expressions involving these operators evaluate to either **true** or **false**. **True** is represented by 1 and **false** by 0. (In the general case, **true** is represented by any non-zero value). If a comparison is against 0 (or falsity), an implicit form of the comparison may be used where the operator and the 0 are omitted.

Other operators which return **true** or **false** are && (logical AND) and || (logical OR). The && operator returns **true** only if both operands are **true** (i.e. non-zero). The right operand is only evaluated if the left operand is **true**. Similarly, the || operator returns **true** if either of its operands are **true** but the right operand is only evaluated if the left

operand is **false**. The operators || and && are associative from right to left.

Bitwise shifts are available in C using the >> (shift right) and << (shift left) operators. Zeros are always used to fill vacant bits when unsigned values are shifted but the value used in signed shifts is dependent on the implementation. Other bitwise operators are & (AND), | (OR) and ^ (EXCLUSIVE-OR).

Two expressions may be concatenated using a comma. In this case, both expressions are evaluated, but the result of the left expression is discarded and the result of the right expression used as the result of the combined expression.

The assignment operator is =. When assignment takes place, truncation may occur if the destination variable is of a lower precision than the source variable. The assignment operator may also be combined with any of the arithmetic or bitwise operators when an operation is required only to transform an operand (e.g. *x* += *y* *increments* *x* by *y* but the expression *x* is evaluated only once).

C also has a conditional operator which takes the form *a* ? *x* : *y* (where *a*, *x*, and *y* are expressions). If *a* is true the entire expression evaluates to *x*. Otherwise, the value of the expression is *y*.

In addition to the binary operators described above, the unary operators in C are * (contents of an address), & (address of a quantity), - (unary minus), ! (Boolean negation), ~ (bitwise complement), (*type*) (change the type of a quantity), **sizeof()** (number of bytes taken to store a quantity, ++ (increment) and -- decrement. The ++ and -- operators may be placed either before the variable, in which case the operation is performed before the value is used, or after the variable, in which case the operation is performed before the value is used. Where

associativity is defined, unary operators are associative from right to left.

The order of precedence of C operators is given in Fig. 5.1. Operators may be grouped in an arbitrary order by the use of parentheses. The order of evaluation of expression operands and function arguments is not defined by C and care must be taken to avoid unwanted side effects when calling functions with complex arguments. In addition, C compilers may arbitrarily rearrange expressions which involve the commutative operators *, +, &, | and ^.

5.1.3 Control Flow and Function Calls

Statements in C are terminated by a semicolon. A group of statements may be grouped together to form a compound statement by enclosing the block of statements in braces ({}). Syntactically, a compound statement may be used wherever a simple statement may be used.

Statements are normally executed in the order in which they appear in the programme source. Branching is provided by the **if** statement and the **switch** statement. Iteration is provided by **for**, **while**, and **do** loops and by **goto** statements. The syntax of these statements is given in the more formal definition of the C language in Appendix 1.

A function in C is a block of statements which is named and is then capable of being invoked from any place in the programme as many times as needed. Each function requires a definition which, apart from the body of the function, also gives the type of the result of the function and the types and names of all formal parameters to the function. If the type of the result is omitted, the compiler assumes a result of type ***int***. The same applies to the types of the formal parameters. If the function returns no value, then its type should be specified as ***void***.

The parameters to a function are always passed by value and no checking is carried out by the compiler to ensure that the types or numbers of the actual parameters match those of the formal parameters.

5.1.4 Arrays and Pointers

Arrays provide a set of indexed variables of the same type. Arrays are declared in the same manner as ordinary variables but the name of the array is followed by a number of dimensions, each enclosed in square brackets (e.g. ***int*** a[*x*][*y*] declares a two-dimensional array containing *x*×*y* elements). The bounds of the array are fixed at compile time and the indices of each dimension begin at 0.

Arrays may be passed as parameters to functions and the bounds of the first dimension of the array may be omitted from the declaration of the formal parameter. Strings are arrays of characters which are terminated by the character '\000'.

Pointers are variables whose value is the address of another variable or of a function. A pointer is declared as being a pointer to variables of a particular type. When pointers are incremented or decremented, the actual value which is added to the value contained in the pointer variable is the size of the object to which the pointer was declared as pointing to.

Arrays are actually constant pointers. When an array is passed as a parameter to a function, the address of (i.e. the pointer to) the first element of the array is the value which is actually passed. Pointer variables may be offset by indices in the same way as elements of an array are referenced, to provide the same results.

5.1.5 Structures and Unions

Structures are used to group a set of variables which are related but are not necessarily of the same type. Once a structure has been defined, the name of that structure may be used wherever the name of a simple type may be used. Structures and variables may have the same names.

C provides an easy syntax for referencing an element of a structure when given a pointer to that structure. Earlier versions of C did not allow direct assignment to a structure but this restriction has been removed in later versions.

Unions are similar to structures and the syntax for declaring and referencing unions is the same as that for structures. The difference is that, whereas each element of a structure has its own storage area, the elements of unions share the same storage area and are referenced as different objects at different times. Where the sizes of the elements of a union differ, storage is allocated to contain the largest element.

5.1.6 The C Preprocessor

When a C programme is being compiled, a preprocessor is always run as the first stage of the compilation to process any source lines which begin with the '#' character. The preprocessor provides constant and macro definition, file inclusion and conditional compilation. Some constants are pre-defined by the compiler (e.g. the type of machine which the code is being compiled for).

5.1.7 The Use of C in Systems Programming

C was written as a successor to BCPL to run on the PDP-11 computer system. The enhancements to BCPL were mainly the introduction of data types and structures. Nevertheless, C compilers do not perform strict type checking and a lot of scope exists for implicit or explicit type conversion. Many of the features of C reflect the type of operations which were available on the PDP-11. Often, such operations form a subset of the operations provided by modern computer systems.

C can therefore be used in two forms. As a high-level language offering a large variety of data types, C can be used in situations where a high degree of rigidity is required in the architecture of programmes. Although this rigidity is not fully tested by the compiler, another programme checker "*lint*" is available as part of the environment of the UNIX system which performs strict checks on compatibility between types, parameters and other features of C programmes.

There are situations, however, when strict type checking is more of a hindrance than a help. When directly accessing hardware elements, for example, the programmer wishes to refer to locations in terms of their physical addresses within the computer system. Using C, such addresses may be used as pointers without generating compiler errors due to incompatibility of types.

Functions in C do not have to be defined before they are referenced. The C compiler therefore knows nothing about the structure of a function which is called (unless it is desired to declare the type of its return value). C functions therefore interface easily to machine code routines and the ability of a C programme to directly access hardware elements and operations may be increased by providing small machine coded routines to augment the bulk of the programme.

The C language itself has no concept of concurrency of routines. Nevertheless, concurrent programmes can be written using C on the UNIX operating system by making use of various UNIX system calls to provide the concurrency features. A version of C - **ConCurrent C**[13] - has been proposed, which includes features for real-time concurrent programming, but it is not yet in common use.

5.2 The Implementation of the UNIX Version 7 System on Darkstar

The implementation of the UNIX Version 7 operating system which was adapted to run on Darkstar was originally written to run on a Motorola EXORMACS system which also used the M68000 processor and other hardware elements which were similar to those used in Darkstar. As most of the UNIX operating system is written in C and is device independent, the task of adapting the operating system to run on Darkstar involved changes to a few areas of the kernel:

1) Changing the address references to those of the Darkstar memory map.
2) Changes to the machine-coded section of the kernel to reflect the capabilities of some hardware elements.
3) The addition of new device drivers for the I/O devices present in the Darkstar system.
4) Adopting a new memory management strategy to cope with the type of segmentation required by the M68451 MMU.
5) Changes to those sections of code which made implicit use of the memory management strategy to correctly use the new strategy.

Areas 1-4 above were all well-defined and changes only had to be made to a small section of code. Nevertheless, having changed the memory

management system to allow the use of the M68451, it was found that many changes had to be made to diverse sections of code, all of which made implicit reference to the assumed memory management system.

Hardware address references were mostly confined to device drivers but some references (e.g. to the timer) occurred in other sections of the code. Nevertheless, all such references were defined as constants using the C preprocessor and were easy to find and change.

The major changes to the machine-coded section of the kernel were in the low-level sections which handled the device interrupts. These vectors and low-level routines had to be changed to correctly reference the Darkstar interrupt structure. The bootstrap procedures also had to be changed in order to correctly call any device initialisation routines which were necessary before the system began operation.

The other machine-coded sections were concerned with kernel activities which could not be programmed in C (e.g. saving internal registers when switching between processes) but which were essentially configuration-independent.

5.2.1 Darkstar Device Drivers for the Version 7 UNIX System

Of the devices used in Darkstar, the timer and the serial interfaces were the same as those used in the EXORMACS system. It was therefore unnecessary to make any changes (other than address changes) to these drivers. New drivers were necessary for the hard disk and floppy disk controllers in use on Darkstar.

The floppy disk driver for the FD179X-02 series of controllers was the first to be written. This driver was later expanded to provide more flexibility and the description given here is of its most general form.

The floppy disk driver allowed the use of up to four drives for any combination of double/single density and double/single sided disks. The available operations were: *a)* setting the disk parameters, *b)* inspecting the disk parameters, *c)* formatting the disk with a user-definable format, *d)* reading a number of sectors and *e)* writing a number of sectors. The disk parameters were set for each drive using the ***ioctl()*** system call in which a structure was passed to the driver containing the required parameters. Some checking was performed by the driver to ensure that the specified parameters were not inconsistent, but no checks were made as to the *suitability* of the parameters. The same structure was passed from the driver to the user programme when an inspection of the drive parameters was requested, again using the ***ioctl()*** call.

In order to format the drive, a different minor device number was used. When this minor device was used as the destination of a write operation, it was interpreted by the driver as a request for a track write rather than a sector write. The user programme therefore supplied enough data for a write to the entire track and, as no checking of the data was carried out by the driver, the user programme had complete control of the format of the track.

For normal block read and write operations, requests were queued if the controller was busy at the time the request was made. Requests were not sorted in any way, so accesses were made in the order requested. At the end of an operation (i.e. in the controller interrupt routine), the state of the queue was checked and, if it was not empty, another request was processed. Requests for multiple blocks were converted into a series of requests for single blocks which allowed the driver to handle disks whose sector size was smaller than the block size of the UNIX system and to act as a character device.

The UNIX kernel issues block I/O requests in terms of the logical block number. The floppy disk driver converted this to side, track and sector numbers according to the current drive parameters. No sector skew or interleave was provided by the driver itself but, as the track format was user-definable, this information could be formatted into the track. To cope with more than one drive, an array of information was maintained by the driver containing the current parameters for each drive and the current head position. In general, the driver would have to perform a *seek* operation before performing a read or write so it was necessary to cater for the possibility of having more than one interrupt for each transaction. A flag variable was kept which indicated the state of the current transaction so that the correct action would be taken by the interrupt routine.

In addition to the routines which provided access to floppy disks as block devices, two routines also existed as a part of the driver which allowed access as a character device. These routines called a generic kernel function which performed various housekeeping duties before calling the normal block device routines. The address of these routines were passed as an argument to the function.

All I/O to and from the disk was controlled by the DMA controller and the code to drive this controller was embedded in the driver thus providing easy error detection and possible correction.

The design of the Marksman T Series Winchester disk controller driver closely followed that of the floppy disk driver and only the differences are discussed below.

The drive parameters for the Marksman disks were not alterable by the user. Instead, it is possible to determine the type of drive which is connected by issuing an **"identify"** command to the drive. This was done

by the driver at system bootstrap time and the drive parameters were set according to the information which was returned. The driver also catered for more than one controller being present and a check to determine how many were actually present was also carried out during the initialisation procedure.

Formatting of the drive was done using an *ioctl()* call. The actual formatting of the tracks was largely managed by the controller so the user programme mainly carried out the error reporting.

Due to some ambiguity as to the order in which sectors were accessed using a multiple sector read or write command, such requests were also converted into several single sector commands. In the Marksman driver, however, requests were sorted by logical block number in such a way that, in the queue of requests, the block numbers were always either steadily increasing or decreasing. As the block numbers were calculated so that blocks appeared in contiguous cylinder groups, this also optimised the seek time needed to handle a queue of requests. Because of the way in which commands were presented to the controller, there was no need to perform separate seek operations so the function of the interrupt routine was simplified.

A tape handling facility was later added to the Marksman driver. Direct tape requests shared the same queue as disk requests but were not sorted. Transparent tape requests occupied a separate queue and this queue was examined when the request was made and just before the driver began a direct request. Transparent commands could therefore be processed in parallel with direct requests.

An SCSI interface driver for the Adaptec ACB-4000 series Winchester disk controllers was also written for Darkstar. Although the internal operation of this driver was very different to that of the Marksman

driver, the functions it provided and the system interface were identical. As the controller used logical block numbers, no conversion to physical sectors was necessary and the sorting algorithm used for the Marksman controller provided the most natural method of organisng the request queue.

Smaller device drivers were later added to the Darkstar system when later enhancements (e.g. the ½Mb RAM board) were added.

5.2.2 The Memory Management System

The EXORMACS system used an additive memory management unit with which segments of memory were defined by providing a starting physical address, a starting logical address and the size of the segment in multiples of 256 bytes. The MMU allowed four such segments to be defined and the implementation of the UNIX kernel on that system used all four segments to provide a separate mapping for the header, text, data and stack sections of the current process. Each segment could be arbitrarily positioned and processes were therefore able to occupy a contiguous area of physical memory, with each section being an arbitrary size.

As the segment mapping was linear, the kernel could access user addresses by knowing only the start location of the entire process area and the sizes of each section. It was therefore unnecessary for the kernel to use any hardware mapping in order to perform accesses to the user area. The hardware mapping of the process header was done to facilitate switching between processes. The process header always occupied a constant logical address near the top of the addressing range. Switching processes involved setting the physical address of this segment to that of the header of another process. As the kernel stack

resided in this segment, this change presented the kernel with an entirely different stack. (The location of the top of each stack is saved in the process header when the process becomes inactive and reloaded into the stack pointer when the process becomes active).

The M68451 MMU is capable of defining 32 segments but the maximum size of each segment is governed by the logical and physical boundaries on which it is positioned. Using the most straightforward way of defining a segment with the M68451, segment sizes are always $2^n \times 256$ bytes where n is an integer. The maximum value which n can take is the maximum value for which $(A \div 2^n)$ yields an integer value where A is the (logical or physical) start address for the segment divided by 256. Thus, if it is required to map a 3kb section of memory, this will take a minimum of 2 descriptors, one of size 1kb in which both the logical and physical start addresses are on a 1kb (or greater) boundary, and one of 2kb in which the start addresses are on a 2kb (or greater) boundary. If it is not possible to position any of these addresses on the appropriate boundaries, then more descriptors must be used to map the segment up to a maximum of 12 descriptors, where each descriptor maps a 256 byte section of memory and can therefore be arbitrarily positioned.

If it were attempted to position an entire process sequentially and contiguously in memory and to map the entire process using an M68451 MMU, then the maximum process size which could be guaranteed to be mapped would be 5¾kb (allowing 2kb for the process header and one descriptor to map the supervisor address spaces). Even if a more cumbersome method were used to handle the process header, (e.g. copying each header to a fixed physical location) and two MMU's were used, (the maximum available on Darkstar), the largest process size would only be just under 14kb. This is clearly inadequate for a general-purpose

operating system, so another memory management system had to be devised in order to use the M68451 MMU.

In order to make the fewest changes to the existing code, it was decided to retain the feature that the process (including the header) was contiguous in memory. What was changed, however, was the requirement that the process be sequential in physical memory. A system was therefore devised wherein the process was fitted into memory in the way which required the fewest descriptors to map it. This meant that sections of code which were logically close together may occupy areas of memory which were far apart and that there was no easy method of determining where any particular section of code would be placed.

In its purest form, this system would have minimised the number of descriptors needed to map a process at a given location but would have also given rise to considerable programming difficulties in the kernel. The following modifications were therefore employed in order to prevent this: *a)* each section of the process was kept together in physical memory so that three separate fittings were used to map the entire process and *b)* the process header was kept physically sequential to facilitate accesses to this frequently used area.

As the kernel could no longer directly access the user area, the information in the user descriptors was mirrored in a group of descriptors which mapped the user area into a logically sequential area of memory near the top of the addressing range. Supervisor accesses to this area therefore used a sequential mapping of the user process.

It is often necessary for the kernel to copy all or part of the user process to another area of memory. With non-sequential physical memory, this becomes a difficult process so another group of descriptors was used to map the new area of memory into the supervisor space when copying

took place. At the end of the operation, the user descriptors and the normal supervisor image was set to map the new area. These "copying descriptors" were also used for certain DMA operations and a locking arrangement had to be used, to ensure that only one process could access them at a given time. Processes which required the use of the copying descriptors, but found them to be temporarily inaccessible, suspended execution until the copying descriptors were freed.

The descriptor allocation used on Darkstar is given below:

MMU 1		MMU 2	
0:	Kernel global mapping	0-15:	User mapping of process
1-8:	Process header mapping	16-31:	Normal supervisor image
9-15:	Spare		
16-31:	Supervisor image when copying		

The segment-fitting algorithm examined each section of code to determine the size of the largest possible segment which could be mapped using a single descriptor. A search was then made in the available address range for a boundary which would accomodate a segment of this size. If such a boundary was found, the segment was mapped into that area of memory and removed from the search list. The procedure was then repeated for the next largest segment. If a convenient boundary was not found, the original segment was divided into two new segments, each of half the original size, and the procedure was repeated using the new size as the target.

The searches were carried out by manipulating two lists in memory. Each list contained linked structures which indicated the size of a segment and the address. One list contained the physical addresses and the other the logical addresses. The process of initialising the lists

involved placing the elements in order of decreasing size, so a comparison between the two lists was easy. When a match was found, the structures describing the segment were removed from the lists and added to a free list by pointer manipulation. When a segment was divided, a element from this free list was inserted into the appropriate list to describe the newly generated segment.

This method of segment-fitting required that the calculations be done whenever a process became the active process and when it was copied to a new area of memory. When the current process was switched, all descriptors were disabled. If it proved impossible to map the process into the allocated memory area using sixteen descriptors or less, this was treated as a fatal error for the entire system.

It was originally intended to use the spare descriptors in extreme cases where this fatal error would otherwise occur. In practice, however, it was found that, following a change to the memory allocation routine, the error situation never developed and sixteen descriptors were always sufficient to map processes which were run on Darkstar. The change to the memory allocation algorithm was made so that, when an area of memory of a certain size was requested, an attempt was made to return an area of memory which was located on the most suitable boundary for mapping that size. Previously, memory had been allocated on a first-fit basis.

Although this method of memory management provided a working system, there were certain drawbacks associated with its use. Firstly, as an extended comparison had to be made each time the process became current as well as on certain other occasions, the operating system lost in efficiency as a great deal of the processing time was spent in locating the process. Also, with the non-sequential physical arrangement, when the operating system needed to know the actual physical address of

an area of the user process, the three ways of doing this (re-running the lengthy mapping algorithm, maintaining a large data area in memory to describe the mapping of the process or making use of the M68451's direct translation facility) were all slow to run. (The method used on Darkstar was to interrogate the MMU).

Secondly, a great many applications programmes (e.g. debugging tools), relied on the fact that processess were sequential in physical memory. These programmes had to be re-written to emulate the kernel's mapping algorithm in order to correctly locate areas of physical memory.

5.3 The Implementation of the System V UNIX System on Darkstar

The version of the UNIX System V operating system which was adapted to run on Darkstar, and later on the SBC was written by Motorola for the EXORMACS configuration and was termed UNIX V/68. Again the major changes necessary for its implementation on Darkstar were the addition of device drivers, and changes to the address and interrupt vector table. Although some versions of EXORMACS used the M68451 MMU, the bulk of the memory management code assumed the use of a simpler, additive MMU like the one described above. Nevertheless, the operating system contained code for a skeletal MMU driver for the M68451 and this was adapted for Darkstar.

5.3.1 Darkstar Device Drivers for V/68

System V exhibits a greater modularity within the structure of the kernel than Version 7 did. For this reason, it was desirable to change the approach used in the Version 7 drivers and to have a separate driver for the DMA controller. Using this driver, other block device drivers could initiate controlled DMA transfers without having the complicated

code necessary to communicate with the DMAC as well as the storage device.

The DMAC driver allowed each device to request transfers of unlimited size in either direction. The DMAC driver handled all error checks, corrections and reporting both before and after the initiation of the transfer. In addition, it could be specified whether the DMAC or the block device interrupt would signal the end of the transfer. (When reading a partial sector from the Marksman Winchester disk, for example, it is convenient to let the DMAC interrupt at the end of the transfer and for the DMAC interrupt routine to call the Marksman interrupt routine. If this is not done, the Marksman controller will never interrupt as the output FIFO will never be emptied and the system will hang up).

The DMAC driver ensured that requests were made to the DMAC for multiple transfers when the requested transfer size exceeded that which could be handled in a single transfer. An internal table specified the type of device connected to each channel so that the correct sort of transfer could be requested for each connected device.

The use of the DMAC driver relieved the block device drivers of a lot of code which handled areas external to the actions of the block device itself. Nevertheless, each driver still had to handle the kernel interface and it was in order to remove this external area that the Darkstar Generic Disk Driver was developed. Drivers for disk devices on the Darkstar system only handled the operation of the specific device for which they were written and this considerably reduced the effort in coding and testing the drivers.

The generic disk driver consisted of a series of functions which mirrored those found in a real device driver. The use of this driver imposed certain restrictions on the internal data structures used by the

device drivers, but once this system had been adopted, the device drivers communicated to the kernel by passing pointers to these data structures to routines in the generic driver.

The generic driver handled the interpretation of the buffer header passed from the kernel, sorting and queuing the I/O requests, sequencing the calls to the disk driver routines, logging errors and signalling the kernel that the transaction had completed. In addition, where an I/O request needed to be performed as a sequence of requests to a particular device for smaller amounts of data, the segmenting of such requests was also handled by the generic disk driver.

The author of a block device driver, therefore, needed very little knowledge of the operation of the UNIX kernel in order to write a driver which would correctly interface to it. This facilitated the writing of device drivers by programmers who knew a great deal about the operation of the device itself but relatively little about the operation of the UNIX system. The use of the generic interface also meant that all devices on the system appeared to operate in the same way which again eased the task of locating a fault in the system.

In a few areas, this could also be seen as a drawback to the system as not all devices would operate optimally when used in the way specified by the generic interface. In practice, the only area in which this was found to be a problem was in the order in which blocks were sorted. To overcome this problem it was made possible to specify that no sorting be performed by the generic disk driver. This meant that the device driver could pre-sort requests before passing them to the generic interface in order to optimise the transfer rate.

The floppy disk driver for V/68 was modelled on that for Version 7 as were the Marksman and Adaptec/SCSI drivers, and used the facilities

offerred by the new DMAC and generic disk drivers. The major difference to the Version 7 driver was that formatting was done by means of an *ioctl()* call rather than by writing to a special minor device. This routine set a flag which marked the transfer as being a track write rather than a sector write.

The Marksman and Adaptec/SCSI drivers provided the same functions as before but the actual coding was much less complex due to the use of the new interfaces. Additionally, a tape handling facility was added to the SCSI driver to provide a backup facility for the disk drives. Character device drivers were also much the same as those on Version 7.

5.3.2 The V/68 Memory Management System

Memory allocation for a process under V/68, as supplied, was done by making three or four separate allocations for the text, data, stack (and shared memory, if used) sections of the process. The process header was included in the allocation for the stack section. This meant that the three sections did not have to be continuous in memory but could be placed where each fitted best.

A table was kept in memory giving the logical and physical start addresses of each section along with the size of the section. Within each section the data was sequential so the search for a physical location first involved calculating which section it was in and then calculating the offset from the physical start address of that section.

Using the M68451 MMU, no attempt was made to map the whole of the process at once, rather, when the process was run it would attempt to access areas of memory which were not mapped and thus cause a bus error. The bus error handling routine checked to see if the access was to a valid location to which no descriptor had been allocated and, if this was

the case, a minimum sized segment would be mapped at that location using a single descriptor and instruction execution continued. If the access location was not valid, the exception was then treated as a genuine bus error.

Descriptors were allocated on a circular basis with the oldest enabled descriptor being used to map the newest required segment if no disabled descriptors existed. At any time, therefore, the current process would have at most 15½kb of its entire memory area mapped. All descriptors were disabled when the current process was switched.

Using this method of memory management, the ability to continue an instruction after a bus error is essential. Because of this, V/68 was not able to be implemented on Darkstar using the M68000 processor. The M68010 processor was used to provide this facility.

Although this method of memory management is less complicated than that used on Version 7, it was found that over 50% of the user time of a process was spent in the bus error routine (i.e. driving the memory management system). This could be improved by using different address space numbers for each process. Using this method, when the current process is switched only the entry in the address space table need be changed rather than disabling all segments and starting the descriptor allocation again from scratch.

5.4 The Implementation of V/68 on The Single Board Computer

After V/68 had been successfully implemented on Darkstar, it was also moved to the SBC. Addresses had to be changed to conform to the Darkstar memory map and the interrupt vector table had to be changed. Calls to the DMAC driver and the internal table of this driver were

changed to provide dual address DMA transfers and the floppy disk driver was changed slightly because of the new controller chip.

Darkstar used the M6850 ACIA's whereas the SBC used the SY6551 ACIA's. The structure of the ACIA driver therefore remained the same although the actual data used to control the device needed to be changed. Additional code was inserted to handle baud rate changing.

The major area of change was in the SCSI interface driver where the Darkstar driver had to be changed in order to drive the M68230 PI/T chip. The strategy adopted was to use the PI/T in a mode where ports A and B were both bidirectional and double-buffered and to provide software arbitration to determine which buffer contained the current status information. There was a potential race in this area as the valid data could be transferred from the alternative data buffer to the normal data buffer during the arbitration process. Nevertheless, it was possible to discriminate between stable and transient data in these registers and therefore to use the correct register at all times, although this involved possible delays in waiting for the data to stabilise. These changes only affected the low-level operation of the driver, so no changes were necessary in any of the more general software which drove the controller.

5.5 Summary

The UNIX operating system is known to be a very portable system[65]. Nevertheless, the three implementations of the UNIX operating system which are described above were a major part of the preliminary work for this project, for the time which elapsed between the original idea of implementing Version 7 on Darkstar and the end of the implementation of V/68 on the SBC was nearly 2½ years.

During this time, the project had been slowed down and complicated by many factors, not all of them technical. As the hardware which was being used was newly designed, the probability of encountering hardware faults was higher would normally be encountered in such an excersise. In the event, quite a few hardware design faults were discovered in attempting to trace faults in the implementation of the operating system.

The major technical complication in the Darkstar implementation was, however, the operation of the memory management units. The development of the eventual memory management system took approximately six months and, after that time, residual faults were continually being found as a result of the high level of interaction between the memory management system and the remainder of the Version 7 kernel.

When it was decided to use V/68 as the basis for this project, new problems arose. Delivery of the original system was delayed by legal problems between the suppliers and the licencers, and by trivial yet time-consuming problems such as the fact that the system, when finally delivered, was stored on different media to that which had been anticipated.

Despite the time lost by such delays, however, the problems of the implementation provided the author with a familiarity with the UNIX kernel which proved to be invaluable in the major phase of this project, and which may have been difficult to attain under different conditions.

Chapter 6

A Real-Time Operating System

The requirements for the real-time operation of processing tasks are mainly in direct opposition to the requirements of a multi-tasking operating system[48]. The major feature of real-time tasks is that the task is *data-driven*. Data must be processed as it is received and cannot be explicitly requested. If the task is not active at the time when the data is available, that data may have been removed or may have become invalid by the time the task is able to process it. Within an operating system, however, the processor is controlled by the operating system task and any other tasks running in the system are allocated processing time according to the criteria of the operating system and not those of the task itself.

Tasks which are required to operate in real time require strict control over the timing of various critical sections of their code. Operations which are performed within these critical sections must be guaranteed to be completed within a certain time-scale and any interruptions to the process during these periods may result in these targets not being met.

Even when it can be guaranteed that a process will not be interrupted within its critical sections, the operations which it must perform while in these sections are often complex and time consuming and the real-time criteria can only be met if the operations are divided so that they can run concurrently. Many operating systems are able to run tasks in such a way that they appear to be running concurrently, but in fact this is normally achieved by interleaving the execution of the tasks and therefore the time taken for the entire operation is not reduced.

Despite the apparent incompatibilty of real-time tasks and multi-tasking operating systems, there are considerable advantages to be gained if it were possible to combine the two modes of operation. Operating systems provide a good environment for the design of programmes. The availability of compilers, assemblers and debugging tools on most operating systems means that tasks may be developed and tested within a known environment which offers a good user interface. Further, many operating systems contain useful software libraries and the use of existing functions from these libraries would reduce the time taken to code and test similar functions for use in the intended task.

Embedded within an operating system is all the functionality needed to perform complex operations, such as disk accesses and spawning of tasks. By utilising such operations, a real-time task can be made much more powerful and flexible without significantly increasing the amount or the complexity of the code within the task. By leaving the system functions to be performed by routines which have been developed and tested specifically for this purpose (i.e. routines within the operating system itself), the programmer of the real-time task can concentrate on problems specific to that task, in the same way as any other programmer on the operating system would develop a normal task.

Having written the real-time programme using the facilities of an operating system, testing its operation would also be greatly simplified if it were to be run under the same operating system. The need for time-consuming transfers between machines would be avoided and error conditions in the programme would be detected and reported by the operating system in the normal way. For example, if the wrong arguments were passed to a function in the real-time task, a bus error may well occur. Without an operating system, the programmer would have to write

(and debug) his own bus error handler so that the condition would be detected. Using the UNIX operating system, for example, the programmer would be informed of this by a message from the shell, and would then be able to employ simple debugging methods to find and correct the source of the error without having to include a large amount of low-level code in his task.

It has been found at the National Aeronautic and Space Administration (NASA) that greater confidence in real-time systems which are to be used in space missions is realised if the tasks have been developed, tested and used under the same system[13]. With this approach, the tasks can be exhaustively tested in the development environment and the problems of moving a task to the real-time environment are virtually eliminated.

In normal operation, real-time tasks could be initiated and terminated in the same way as normal tasks are run on the operating system and routines which specifically handle this user interaction need not be programmed into the task itself. It is also helpful to be able to easily monitor the operation of a real-time task without affecting its speed of execution. Again, under an operating system, low-priority monitor tasks could run on the same system and efficiently communicate both with the real-time task and with the outside world using the facilities of the operating system.

It can be seen, therefore, that a real-time task would benefit in many ways by running in the environment of an operating system, rather than as a standalone process. An operating system which can support real-time execution would reduce the interference to user tasks by system tasks to an insignificant level and therefore *all* user tasks in the system would benefit from this. In addition, there are tasks which, while

not requiring real-time operation, still use a lot of processor time in their execution. These tasks could be treated in the same way as real-time tasks in order to make the operation of the whole system more efficient.

In designing an operating system which would support the execution of real-time tasks, care must be taken that this is not done in such a way as to make the execution of ordinary tasks unduly slow. One of the principal advantages of running real-time tasks under an operating system is that the same environment can be used for both real-time tasks and tasks of lower priority. Other users of the system should therefore be unaware that some tasks are operating in real-time, and therefore the response of the system to ordinary tasks should not suffer any significant degradation during those periods when real-time tasks are executing.

6.1 Real-Time Tasks Under the UNIX Operating System

The UNIX operating system is designed as a time-sharing system. User processes must share the available processor time with each other and also with the system task(s). Any user task which is running is therefore liable to be suspended if a system activity needs to be performed. For example, a running process (*A*) which requires a file to be read from disk makes a call to the system to do this. When the I/O request is placed in the device queue, that process is suspended until the transaction is completed and another process (*B*) is allowed to run. The new process may be suspended at any time, however, when the device issues its completion interrupt, for the kernel must then handle the interrupt. This involves manipulation of the device queues and possibly initiating another disk transfer. Process *A* is then placed in the queue

of running processes. After this is done, process *B* may be allowed to continue execution provided the rescheduling flag had not been set due to a clock interrupt.

Many of the UNIX system activities are lengthy and indivisible. Thus, if a user types the interrupt character at a terminal, the kernel searches through the entire process table and sends the interrupt signal to all processes which were initiated from that terminal. During this time, all user processes are suspended. An attempt to run a real-time process under the UNIX system would therefore fail, as it is impossible to determine when the kernel will suspend the task in order to perform a system function.

Even when there are no other essential activities to be performed, a clock interrupt occurs once every 0.02 seconds. In processing this regular interrupt, the kernel resets the hardware clock. It then checks if any process has requested a callout (a function to be called after a specified time) and manipulates the queue of these functions. After this, the time indications for the running process and other system activities are adjusted. The system's internal representation of the current time is also incremented. If the specified number of clock interrupts have occurred since the last process switch, the rescheduling flag is set and more housekeeping activities are performed including profiling, adjusting the priorities of running processes and handling processes awaiting alarm signals.

If the rescheduling flag has been set, the current process is placed in the queue of running processes and the scheduling process is run. The scheduler attempts to swap in any processes which have been swapped out and searches for the highest priority runnable process - which may be

the process whose execution was suspended in order to run the scheduling task.

Even if it were possible, therefore, to ensure that a real-time process were the only active user process on a UNIX system (perhaps by running the system in single-user mode), the execution of the process would be suspended for a short time fifty times per second, and for a longer time once per second while the system performed the activities mentioned above. Therefore, even on a dedicated UNIX system real-time operation of user processes cannot be achieved.

The other possible requirement for real-time operation which was mentioned above is concurrency of operations. Under the UNIX system, true concurrency is only achieved between two processes if they require operations which do not use processor time. While one process is awaiting a disk transfer, for example, the other process may use the processor as, all the operations connected with the disk transaction are being performed by the disk controller and the DMA controller. Concurrency in real-time tasks is needed however, when there is a high demand for processing time and each process is attempting to execute a processor intensive operation within a certain period of time. Apparent concurrency is easily programmed under the UNIX system by using the *fork()* system call. This call creates another process which appears to run concurrently with the parent process but which, in reality, only runs when the parent process is suspended. There is therefore no advantage in attempting to use the *fork()* system call to achieve an overall decrease in the time taken to perform an operation.

6.2 Limiting Factors in Current Computer Systems

The problems, outlined above, in attempting to run real-time tasks on any operating system and on the UNIX operating system in particular, all stem from the limited processing resources available in most computing systems. The major "building-blocks" of the computer systems available to this project were microprocessors, random access memory, I/O ports and mass storage devices. These components limit the operation of the UNIX system in different ways. The number of I/O ports limits the number of peripherals which can be attached to the system and therefore the *rate* at which a number of users can develop programmes, obtain output, etc. The capacity of the mass storage devices limits not only the amount of information that can be stored, but also the number and sizes of processes which can be run, since sufficient swap space must be available to prevent a deadlock of the system.

The amount of random access memory also limits the size and number of active tasks and also the speed at which they appear to execute, since an inability for a process to achieve the allocation of memory which it requires will result in one or more processes being swapped out to disk. When a process is swapped out it affects the execution times of all other processes in the system, for the operating system continually tries to swap it back in and may swap other processes out to achieve this.

The choice of microprocessor affects the speed at which operations may be performed but, more importantly, the use of only one processor per system introduces a bottleneck which limits the activities of the system. All processing within the system must be done by the single microprocessor. This includes scheduling, I/O, housekeeping activities, and user processes. Some systems alleviate this problem by having separate I/O or front-end processors which perform some of the system

activities, but user processes still experience the bottleneck by having to share processor time with other user processes.

Of the four basic components of the system, it can be seen therefore that the availability of I/O ports would be a disadvantage but would not seriously affect the execution times of processes. The capacity of mass storage devices has some bearing on the overall speed of the system, but the main limitations in a system are the availability of memory and the number of processors.

The use of more memory would reduce the need to swap processes out and therefore processes would experience fewer interruptions but the bottleneck caused by the processor limitation would still be there. The major limitation on the power of a computing system is therefore seen to be the architecture of such systems which is centred around one (or few) processors. If it were possible to have a system which treated processors as another resource which could be allocated to processes in the same way as present systems allocate other resources (e.g. memory) to processes, a significant increase in processing power and speed would be achieved.

Fortunately, the trend in recent years has been the steady fall in the cost of processors. Mass storage devices are still relatively expensive but high capacity memory chips and powerful microprocessors are now readily available and relatively inexpensive.

It is therefore financially feasible to design a computer system which contains many processors and a large amount of random access memory but which only contains a modest amount of mass storage media. Obviously, if the disk capacity is too small, no advantage would be gained by this as it would be impossible to store enough information to fully utilise the processing power of the system and many processors and

areas of memory will be inactive for a large proportion of the time. Nevertheless, if sufficient disk capacity is installed which would be adequate for an existing single-processor multi-tasking operating system, the addition of more processors and memory to such a system would remove the processing bottlenecks and allow fast and efficient execution of tasks.

6.3 The Design of a Multi-Processor Operating System

The UNIX V/68 kernel processes share the same processor as the user processes. This is the first significant bottleneck in the system. For example, on receipt of an interrupt signalling the completion of a disk read operation, the system must activate the task which requested the transfer, copy the data from the system buffer into the user data area and allow the task to continue in user mode. In practice, however, another user task may be running when the interrupt occurs. The system must then allow this process to continue until it is preempted or waits, before it can copy the data from the buffer.

In attempting to relieve such bottlenecks, it would therefore be helpful if a number of processors in the system were dedicated to performing system tasks only. This would allow system activities to proceed in parallel with user tasks which do not require system facilities at the time.

Conversely, the execution of user processes is also adversely affected by system activities and the activities of other user tasks. A task which enters the kernel is not preempted until it reaches a stage where it can no longer proceed (e.g. waiting for I/O) or until it is about to resume normal execution in user mode. During this time, all other user tasks are suspended.

If a number of processors in the system were dedicated to running user tasks only, a number of user processes could be executing while the system processors dealt with those processes which have made calls to the system. In addition, for a given time period, each user process would receive a larger amount of processing time as there would be a greater chance of a processor becoming idle.

While it is clear that the availability of a multiplicity of processors (which are able to run user tasks) would enhance the performance of a UNIX system, there has been some discussion over whether these processors should all be allowed to run the task in system mode as well, or whether system activities should be handled by separate processors. *Bach and Buroff*[2] argue that, as most normal tasks spend about 40% to 50% of their time operating in system mode, restricting a particular processor from running in system mode prevents the system from achieving its full potential *except for specific workloads.*

For this project, however, the aim has been to maximise the performance of a very restricted range of tasks i.e. those which are to be run in real time. In a real-time task there is the necessity of maintaining strict time constraints on many sections of code and such tasks tend to spend less than average time in system mode. It is therefore desirable to free such tasks from any possible interference from unrelated system activities.

In the system being designed, therefore, there will be a rigid division of activities between processors. Some processors will exclusively execute system tasks and user tasks which are running in system mode. All other processors in the system will exclusively execute user tasks which are running in user mode. This not only achieves the above requirements, but also reduces the areas of contention which arise when

there are multiple accesses to shared data (e.g. system tables) for diverse uses. Systems which allow all processors to run tasks in system mode require extensive locking mechanisms to preserve the integrity of system data and such mechanisms must be used throughout the entire system. It will be shown that the scheme adopted for this project need only employ a simple, localised locking mechanism.

The allocation of the system processors will be related to the performance of specific system activities (e.g. I/O processors), whereas user processors will be allocated to user processes according to an algorithm which ensures efficient use of the processors across processes with differing priorities and modes of operation.

When a task is run on the system, it will be allocated to a particular processor and will be loaded into the area of memory associated with that processor. This will be done by a system processor which will then signal the user processor to begin execution of the task. The user processor will ensure that its memory management system is set up in accordance with information which is passed to it from the system processor and will then commence running the user process in user mode.

At this point, the user processor will, in effect, be dedicated to the user task and will not attempt to suspend its operation until one of the following two conditions occurs: *a)* an exception is generated by the user process or *b)* a signal arrives from a system processor.

If an exception is generated by the user process, the user processor will signal the system processor which handles such an exception and will suspend all activity until a new signal arrives from one of the system processors. It should be noted that one important type of exception is that which is generated when the user process makes a system call. As *all* exceptions will be handled by the system processors,

this fulfils the condition that all system activities will take place on the system processors.

Whenever a signal arrives from a system processor, the user kernel will begin the execution of the specified process. Signals from the user processors to the system processors therefore occur when a user process requires a system operation. Signals from the system processors to the user processors occur when it is time for the user processor to reschedule. If the user processor is currently running a process when a signal arrives, it must first save the context of the current process before attempting to restore the context of the new process.

If the system is currently engaged in an indivisible activity when an interrupt arrives from a user processor, it must save the information pertaining to that signal and queue the request for later action. Before resuming its interrupted activity, however, it must signal the user processor to execute another process if one is waiting. In this way, processors are idle for the minimum amount of time.

Like processor allocation, scheduling of processes is performed by one of the system processors. This processor will handle the 50Hz clock interrupt and perform all necessary housekeeping activities relating to this signal. When the time for preemption arrives, this processor must ensure that there is another process to be run, before signalling the relevant user processor. In this way, user processes are only interrupted when it is absolutely necessary to do so.

6.4 Real-Time Processes in a Multi-Processor Environment

The system proposed above provides an excellent environment in which to run real-time tasks and provides the opportunity for true

concurrency of operation as two processes can run on different processors and can therefore be running at the same time.

The problem of preemption is easily solved using this system by the addition of another system call to the UNIX kernel. This system call will inform the operating system that the calling task requires to be run without interruption from the system. On receipt of such a call, the operating system will attempt to allocate that process to a processor of its own. If such an allocation is not possible, the system call will fail and this will be taken as an indication that the system is too busy for such a process to be initiated.

If the system call succeeds, all other processes will have been removed from the allocated processor and no further processes will be allocated to this processor until the real-time process has terminated. As the processes are only interrupted when it is absolutely necessary to do so, such processes will never be preempted by the system as there will never be another process which is available to be run by that processor. Real-time operation is therefore made possible by the addition of this system call.

Communication between concurrent processes is achieved by the normal UNIX inter-process communications methods. Thus, under System V, concurrent processes will be able to communicate with each other by means of pipes, signals, shared memory, semaphores and messages. The operating system will handle the fact that the processes are actually running on different processors, and this will be transparent to the programmer. The use of shared memory segments will provide the real-time process with the greatest measure of detachment from the activities of the kernel since, once these segments have been set up, data will be transferred between them by the user processor without access to

the system. If any of the other methods of communication are used, the process must make a system call for each transaction which implies that the user processor will suspend the execution of the process until the return of the system call. Such calls cannot, therefore, be made from within the critical sections of the task.

Where communication by shared memory is not available (e.g. disk transfers or terminal I/O) a system call must be made. If this conflicts with the operation of the real-time task, however, support tasks may be used in order to release the real-time task of the time-consuming communication with the operating system. The support tasks would be tasks of a lower priority which would (necessarily) be running on other processors. Communication between the real-time task and the support tasks would be via shared memory to eliminate the system overhead. One function of the support tasks could be to read the data which the real-time task requires for an I/O transaction and to make the call to the operating system which would perform the actual I/O.

By using this method of I/O, the real-time task never needs to communicate directly with the operating system and would never be in danger of having to wait while the operating system dealt with its request and others.

Support tasks could also be used to provide monitoring facilities to the real-time task. In this way, an operator could monitor and communicate with the real-time task in a controlled way which would not degrade its speed of execution.

Where I/O needs to performed in real-time, the use of system calls (e.g. *write()*) would not provide the necessary level of performance. In order to perform its own I/O, a task would need to be able to access the

I/O page of the system and, if interrupts are used, to trap interrupts and provide its own interrupt handler.

Access to I/O registers requires a modified form of shared memory operations, where the memory segment which is allocated to the process is actually known to exist in a specific physical location i.e. the I/O page. The ability to trap interrupts can be provided by a modified form of the *signal()* system call, where the specified signal is a known interrupt vector. The normal value for that vector would be replaced by the address of the specified handler or the address of a small section of code which would perform essential activities (e.g. saving processor registers) before calling the specified handler.

6.5 Summary

Although it can be seen that there are considerable benefits to be gained by running real-time tasks in the environment of a multi-tasking operating system, it is difficult to combine the two concepts using a single-processor system due to the number of activities which need to be performed by the operating system and the time constraints on the real-time task. The cost of processor and memory chips make it feasible, however, to design multi-processor systems which offer an increase in processing power though maintaining the same mass storage capacity.

An operating system can be designed for such a computer system, using the UNIX system as a basis, which will work by running system tasks on some processors and user tasks on others. This configuration relieves the major processing bottlenecks in the performance of the operating system.

At first sight, this appears to achieve a very small increase in processing power for a significant increase in hardware, since the

minimum system will require two processors (one system processor and one user processor) to achieve the same functionality as a normal single-processor operating system. However, even a two-processor system allows concurrency between system tasks and user tasks and thus achieves some improvement in the overall execution of user tasks but, more importantly, more user processors may be added to the system which would significantly increase the processing power and flexibility of the system to magnitudes which are unattainable with single-processor systems.

Using such a system, it would be possible to assign tasks which require real-time execution to specific processors to which no other tasks are allocated. If the operating system only interrupts the task when absolutely necessary then real-time operation can be achieved by this method.

Such tasks could be programmed as normal tasks on the UNIX system and their mode of operation will be transparent to the programmer and to other tasks within the system. True concurrency of operation could be achieved by this method, and communication between real-time concurrent tasks would be achieved using the normal UNIX inter-process communications facilities. Where normal communication with the operating system would endanger the real-time execution of the task, support tasks may be used to isolate the real-time task from the operating system or generalisations can be made to some system calls to provide the task with better access to the physical machine.

By using such a system it would be possible to easily develop and execute large and complex real-time tasks which exhibit a manageable user interface. The main goal of this project has therefore been the implementation of a multi-processor, multi-tasking operating system as described above.

Chapter 7

The Implementation of

A Multi-Processor Operating System

The description of the multi-processor operating system given in chapter 6 is a generalised view of what is possible, given existing technology. The task of implementing such an operating system requires that certain decisions be made regarding which features of the general description are most important and should be strictly adhered to, and which features are of lesser importance and can be regarded as future enhancements to the basic system.

The basic decisions which were made for this project was that the UNIX System V operating system should be used as the basis for the work, and that the resultant operating system would be implemented on the SBC computer system described above. Having determined the basic hardware and software, it was then possible to make decisions as to how to approach the problem of achieving the specification for the real-time operating system.

Despite the need to concentrate on a fairly narrow hardware and software configuration for the purposes of this project, it was felt that the resultant operating system should still be made to be as portable as possible. The UNIX system achieves its high portability by extensive use of a high-level language and the isolation of machine dependencies into well-defined modules. The UNIX system approach would be used in the system under development.

In order to reduce the time which would need to be spent in exhaustive debugging of the system, it was decided that the changes to the existing V/68 system would be kept to a minimum. This would allow

development to take place in an environment where it was known that the majority of the code had already been tested and was in working order. Coding inaccuracies could then be more easily pinpointed as they were almost certain to occur only in the modules under development.

In addition to minimising the number of changes to the existing kernel, it was felt that no change should be made to the UNIX user interface. One of the strengths of the UNIX system is its very simple, flexible and straightforward user interface described in chapter 4. Many application programmes have been written using this interface and it would therefore be advantageous if the real-time operating system appeared to be the same as the standard System V UNIX system.

This guideline has had to be relaxed in one area in order to fully implement the system. To be able to guarantee real-time operation, the operating system must be informed when a real-time task is present so that it can dedicate a processor to it. This necessitates the addition of one new system call to the system but, as this is merely an addition to the system which would only be used by real-time processes and as it does not interact with any other area of the user interface, the normal user interface remains unchanged.

The general specification for the multi-processor operating system envisages a strict division of processors in the system into system processors and user processors. Due to limitations to the availability of hardware for this project, it was decided to have an effectively unlimited number of user processors, but to limit the number of system processors to one. As the aim of this project is to enhance the execution of real-time user processes and, as these processes would spend only a minimum amount of time in system mode, it was felt that this was not an undue restriction on the hardware configuration. Later, measurements can

be made to determine which areas of the kernel would benefit from being run on a separate processor and the system can be divided accordingly.

Having only one system processor also reduces the programming effort required to implement the first version of the system. This cannot, however, be extended to allow the development of a system with only one user processor in order to achieve a shorter development time as real-time operation is only possible where there is more than one user processor in the system.

Each processor board in an SBC system contains both a processor and a quantity of local memory. In the description which follows, data will be described as being located on a particular processor. This indicates that the data is located in the local random access memory which is situated on the same board as that processor and which that processor accesses at memory locations beginning at 0. The terms "processor number" and "board number" will also be used interchangeably and a process will be described as having a certain processor number or board number if it is located on the processor board which has this number in its processor number register.

7.1 General Features of the Implementation

V/68 maintains the following information about processes running on the system: *a)* a process table which contains information about the priority, permissions and state of the process, *b)* an MMU table which contains the logical and physical addresses of the four sections ofthe process, *c)* a text table which contains additional information about the state and location of the text segment of processes using shared text and *d)* a process header which contains local information about the process and which can be swapped out with the process. These are the main

system tables which identify the state and location of the process. (The process headers are actually not arranged in a contiguous table but reside in the process's stack segment). The tables are cross-referenced so that all the information pertaining to a process may be accessed once the location of one field of information has been found.

To implement a multi-processor version of the operating system, the following changes were made to the system tables:

a) The process headers were no longer stored with the process, but in a table in the memory area of the system processor. As the header information is referenced frequently by the system processor (the system stack resides in the process header), but only rarely by the user processors, this arrangement reduced the number of backplane bus cycles which would be required in accessing the headers and ensured that the system stack resided on the system processor. Using this arrangement, the process headers were no longer swapped out with the process but, as they occupied no user memory space, this was not felt to be a disadvantage. The process table and the process header table could, in fact, have been combined into one, but this would have required unnecessary changes to the kernel.

b) The process table, MMU table and process header table were arranged by slot number. The slot number is the index of the required record in the table. This removed the problem of cross-referencing the tables and meant that, by passing only the slot number to the user processor, all the relevant tables could be referenced directly.

c) One addition was made to each entry in the process table to hold the processor number of the user processor running the process. This number is used to index the processor table.

d) The elements of the text table were expanded to include an indication of which user processors had copies of a particular shared text segment.

e) A new table was added to contain information about the user processor. This table contains a flag variable which indicates the state of the user processor, the slot number of the running process on that processor, the slot number of the next process to be run on that processor and pointers to the memory map of the processor and to the queue of running processes on that processor (the "runq"). To facilitate the allocation of processes to processors, the processor table also contains the number of loaded processes, the number of processes on the runq, the amount of free memory on the processor board and the number of clock intervals that the processor had been idle.

f) An array was added to the process header in which the registers of the user processor were dumped when the execution of the process was suspended. There are three such arrays in the standard UNIX process header but this new array is somewhat larger than those as it contains all the relevant information from the exception stack frame. The elements of the new array are, in fact, the arguments to the kernel's generic exception handling routine.

The general specification of the multi-processor system is that the system processor contains the kernel and performs all of the system functions, while the user processors contain only the user processes and

perform no system functions at all. This would require the system processor to be able to load the internal registers of the user processors and to set up the memory management units of the user processors. This is not possible using the SBC system as the local I/O page of each processor is inaccessible by other processors. The specification also requires that the system processor be signalled when an exception occurs on a user processor. Without allowing some kernel functions to be performed by the user processor, this would also be impossible on the SBC system.

In practice, therefore, the user processors must perform some system functions but, to remain as close as possible to the ideal specification, these should be as few as possible. The approach which was adopted for this project was to have a skeletal kernel running on each user processor. This user kernel was given the mnemonic **VRK** (very residual kernel). The functions of the VRK were those which could not be performed by the main kernel on the system processor in the SBC system.

The VRK handles all exceptions using one exception handling routine which dumps all its internal registers onto the system stack, copies the resultant stack frame (including the exception stack frame) into the area reserved for it in the process header and signals the system processor. Execution of the current process is suspended and the user processor idles until a signal arrives from the system processor.

Inter-processor signals in the SBC system are interrupts which use the level 5 autovector. A signal from the system processor would therefore cause the VRK to enter its exception handling routine and save the stack frame including the register dump. A signal to the user processor results, therefore, in the context of the current process being saved.

The actual operation of the VRK's exception handler is not quite as homogeneous as described above as two special cases are recognised and acted upon directly by the user processor. The first of these is the above-mentioned inter-processor signal. The VRK recognises this as a signal from the system processor for it to begin the execution of a new process. After the stack frame has been saved, therefore, the VRK examines the processor table to see which slot is indicated as the next process to be run and loads the user processor's internal registers with the data in the array of the new process header. Execution of the new process then begins.

The second special case is a bus error while executing in user mode. The VRK uses the same memory management technique as the normal V/68 system so such a bus error may be an indication of a page fault. If this is the case, the stack frame is not saved in the process header. Instead, the VRK sets up a new MMU descriptor to map the required segment and continues execution of the process.

The actions of the VRK are therefore *a)* to detect exception conditions and to notify the kernel when such a condition occurs, *b)* to save and restore the process context and *c)* to handle the hardware memory management of user processes. When implemented on the SBC, the entire VRK (including the exception vector table [1kb], data areas and a generous stack allowance) occupied less than 2kb. (The standard V/68 kernel as implemented on the SBC occupies well over 100kb and since SBC boards are populated with a minimum of 256kb of RAM, the inclusion of the VRK is barely significant).

If it could be guaranteed that only one process were to be running on a user processor, the requirement for the VRK to handle page faults could also be relaxed. In a multitasking environment, the siting of the

VRK at the bottom of memory means that no memory management is necessary for the VRK, since its logical addresses are the same as its physical addresses. This leaves the maximum number of descriptors free to map user processes.

The VRK could, however, be sited at the top of memory, in which case the physical addresses would have to be mapped by the MMU into logical addresses at the bottom of memory and the vector base register of the M68010 would be set to the base of the VRK, ensuring that exceptions are trapped correctly.

This scheme would also require memory management for user processes in general, but one user process could be placed at the bottom of memory, in which case its logical addresses would be the same as its physical addresses and its address space would not need to be mapped by the MMU. Whenever this process generated a bus error, therefore, it would be be a genuine bus error and not a page fault. The bus error exception could be passed to the kernel in the same manner as all other exceptions. If such a process were the only running process on a user processor, the VRK would therefore be relieved of handling page faults, thus reducing the system overhead on the user processor.

It will be shown that, when running real-time processes, the situation will arise where a process is guaranteed to be the only active process on a particular processor. Such a proposal for the location of the VRK should therefore be considered, in order to reduce the system overhead in such cases and thus enhance the execution of the real-time task.

The SBC system supports only one type of inter-processor communications. It can be seen from the above description of the VRK, however, that this is adequate for the purposes of the multi-processor

operating system. A signal from the VRK to the kernel indicates that an exception has arisen on the user processor and that it should be handled by the kernel. By examining the processor table, the kernel determines which process was running at the time of the exception and can therefore locate the exception stack frame in the process header. By simulating this stack frame on the system processor, the kernel can then run the user process in system mode as normal.

A signal from the kernel to the VRK indicates that the user processor should suspend the current task (if any) and begin execution of a new task. The slot number of the new task is entered in the processor table before the signal is sent so the VRK can locate the context in the appropriate header. (In practice, the VRK may be signalled to begin no new task at all. This is done because of the locking arrangement used in the VRK and is covered below).

When a process is to be started, a processor is allocated to it according to an algorithm described in section 7.4.1. The process image is then copied to that processor. This procedure is the same as that used in the single processor system as the destination of the process image is given in the MMU table. Because of the global addressing capabilities of the SBC system, the processor number determines the address of local memory for that processor so the value which is entered in the MMU table is directly related to the processor number of the process.

After the process image is installed on the user processor, the process is entered in the runq of that processor. Again, this procedure is the same as in the single-processor system except that each processor (including the system processor) has its own runq. When the process is to be run, it is taken off the runq and its slot number is entered in the

processor table entry for its processor. A signal is sent to the VRK of that processor and the context of the process is loaded as described above. When the process is preempted it is replaced in the runq as normal.

If the process makes a system call or raises another type of exception (other than a page fault) the VRK saves the stack frame and signals the kernel. The process is then placed on the runq of the system processor and is scheduled in the normal way. (Before continuing normal processing, the kernel tries to schedule another process on the now idle user processor). If the execution of the process is not terminated (e.g. as it would be if the call was the *exit()* system call) while running in system mode, the process will be replaced on the runq of the user processor when it returns to user mode and will be scheduled for the user processor as normal.

In the case of normal processes, it is not guaranteed that the process will reside on the same processor after its execution is resumed. While running in system mode, it may have been swapped out to disk and allocated to another processor when being swapped in, or it may have been copied directly to another processor for reasons of system efficiency. These actions are, however, transparent to the process, and the process is never aware of the identity of its processor. Real-time processes *are* tied to a specific processor and always reside on that processor until terminated.

7.2 VRK Structure and Memory Management

The code for the VRK consists of the interrupt vector table, 6 assembly language routines, 5 C routines, 5 global variables and a number of local variables. The executable version of this code resides in

the file "/etc/vrk" and, as with the normal bootstrap file "/etc/init", the operating system will not enter its normal operational mode if this file does not exist or is not marked as being executable.

The interrupt vector table is loaded as the first module of the file. Immediately following this table are the global variables. Four of these are initialised by the kernel and contain the processor number of the system processor, a pointer to the user processor's entry in the processor table, the address of the start of the MMU table and the address of the start of the process header table. The fifth global variable is used to implement a locking mechanism and is described below. The global variables are accessed by the kernel using the following C structure:

```
#define NVEC 255  /* Exception vectors excluding initial sp */

struct vrk
 {
      int k_lomem;                        /* top of VRK stack */
      int k_vec[NVEC];          /* remainder of vector table */
      int k_lock;                            /* lock variable */
      int k_super;              /* number of system processor */
      struct slt *k_vp;               /* user processor entry */
      struct mmu_table *k_mtp;                   /* MMU table */
      struct user *k_up;              /* process header table */
 };
```

where the structures **slt**, **mmu_table** and **user** are defined in global header files and describe the entries in the processor table, MMU table and process header table respectively. As the exception vector table for each processor begins at local address 0, the address of this structure is easily found for a given user processor as it is the global base address of the of the processor board: (processor number)$\times 40000_{16}$.

As a part of the system bootstrap procedure the memory maps of each processor which is present in the system are initialised. (The

presence of a processor is determined by the ability to read its control register). Memory maps are arrays of structures containing the start address and size of free areas of memory and are manipulated by the kernel functions ***malloc()*** (allocate memory) and ***mfree()*** (free memory). Memory on external memory boards is included in the memory map of the system processor. (The function ***malloc()*** takes a processor number as one of its arguments, thus ensuring that any memory which is allocated resides on a known board).

After the memory initialisation, the processor initialisation takes place. The file "/etc/vrk" is checked and read into the memory of the system processor. If this file does not exist or is not executable, the system loops at this point and does not continue the bootstrap procedure. The size of the executable task is rounded up to the nearest convenient boundary after a suitable stack allocation has been added and this value is written into the first four bytes of the process image (**k_lomem**) as the intial stack pointer value. The processor number of the supervisor processor, the address of the MMU table and the address of the processor header table are also inserted into the appropriate places in the process image (**k_super**, **k_mtp** and **k_up**).

For each processor in the system, the address of its entry in the processor table is inserted into the process image (**k_vp**) and the memory allocation function is called to provide enough memory on that processor for the entire process from the beginning of the exception vector table to the value given by **k_lomem**. As ***malloc()*** returns segments on a first-fit basis, the area of memory at the base of the processor will always be returned and the memory map will be changed so that subsequent allocations (for user processes) will occur above the VRK. The image of the VRK is then copied to the user processor and a reset exception is

generated on that processor by clearing bit 5 in the processor control register (see chapter 3).

The reset exception will cause the user processor to load its programme counter and stack pointer with the values in the first 8 bytes of its local memory and to begin processing at the location in the programme counter. The value in the stack pointer is set by the system processor and puts the stack pointer at the top of the memory area allocated for the VRK. The value in the programme counter is set when the process is linked and points to the first of the machine code routines "*start:*". This routine masks all interrupts so that no signals from the system processor are received before the initialisation is completed. It then clears the stack area and the unitialised data area and calls a C routine ***main()***. The ***main()*** routine initialises the local variables and calls the routine to initialise the address space tables in the MMU's. In the first implementation, a single address space number 0 is used for the system address space, and a single address space number 1 is used for all user address spaces.

After this initialisation, the assembly language routine ***wait:*** is called. This routine is used throughout the operation of the VRK and is unusual in the fact that it has no return path. When ***wait:*** is called, the stack pointer is reset to the top of the stack area and the **stop** instruction is executed. The processor will therefore idle until an exception occurs after which it will execute the instruction immediately following the **stop** instruction. In the case of the VRK, the instruction immediately following is a branch to the beginning of the ***wait:*** routine.

The normal state of the user processor is, therefore, to idle. As the stack pointer is reset at each invocation of ***wait:*** the VRK maintains a very limited history of its operations.

Apart from the initial stack pointer and programme counter values, all entries in the exception vector table point to one routine ***handler:***. This routine is called for all types of exceptions and begins by masking all interrupt levels to prevent the occurrence of ambiguous situations. Unless there is a fault in the coding of the VRK, the only exception which could possibly be generated at this time would be a scheduling signal from the system processor and the processing of this is therefore delayed until the current exception has been processed.

After setting the interrupt mask, the internal registers are saved in the same order as is done in the kernel. At this point, therefore, the stack frame of the user process is exactly the same as would be produced in the single-processor system. The C routine ***vrkintr()*** is then called with the same argument structure as the kernel's exception handler ***trap()***. This routine will only return if the interrupt was the result of a page fault, in which case the stack frame would be discarded and normal processing resumed. In all other cases, the normal exit route of ***vrkintr()*** is to call ***wait:*** after signalling the system processor. Instruction execution is therefore suspended until the next exception is raised.

On entry to the ***vrkintr()*** routine, the VRK checks to see if it is currently running a process. If this is so then it could be that the exception was the result of a page fault. The function ***mmuintr()*** examines this possibility and returns successfully if this is the case, having set up a new descriptor. Normal instruction execution then continues. If ***mmuintr()*** fails, the exception stack frame is saved in the process header by the assembly language routine ***save:***.

Whether the exception occurred while running a user process or not, the type of the exception is examined to see if it was an inter-processor

interrupt. It is assumed that this can only come from the supervisor processor and is a signal to reschedule. As the context for the current process will have already been saved, the VRK reads the slot number of the next process from its processor table entry, uses it to update the current slot number, disables all user descriptors and loads the context of the designated process using the assembly language routine *resume:*.

Under certain conditions described below, the kernel may enter the value 0 in the processor table as a slot number. As the process in slot 0 is the system scheduler, this is taken to be an indication to run no process at all and, in this case, *wait:* is called.

Any other type of exception is not handled by the VRK. When such an exception occurs, the assembly language routine *signal:* is called to send an inter-processor signal to the system processor. The main function of *signal:* is to clear bit 5 of the system processor's control register and to set a flag in the processor table to indicate which user processor sent the signal. In practice, however, other operations must be performed to avoid race conditions. A fuller description of the races involved is given below.

The two other assembly language routines *save:* and *resume:* simply copy the internal registers from the process header to the user processor and back again. *Save:* returns to its calling routine while *resume:* performs an RTE instruction and returns to the user process. Again, the actions of *resume:* are complicated by the need to avoid races.

The routine *mmuintr()* is basically the same as the standard V/68 page fault handler. As the addresses in the MMU table are global addresses, the VRK's routine converts them to local addresses by subtracting the base address of the board before loading the MMU. Shared

memory segments need not reside in local memory, so this conversion is not done for descriptors defining shared memory segments.

Memory management on the system processor is much simpler as no user processes are run there. The only dynamic mapping on the system processor is the mapping of the current process header (and thereby the system stack) to a constant logical address. The process headers occupy 2kb and, in this implementation, the current header is mapped to logical address $FFF800_{16}$ so that it occupies the highest 2kb slot in the processor's logical addressing range.

As the system stack must occupy the same address space as the other system data areas, and as the system processor must access the entire addressing range in order to globally address user processors, a collision would occur if the system processor were to attempt to load a descriptor to map the system stack to $FFF800_{16}$ as this logical address is mapped by descriptor 0 when the system is booted. The system bootstrap procedure therefore sets up a new address space with address space number FF_{16} and maps the entire logical address range *except* the last 2kb into this address space. As each segment must be an integral power of 2 in size, the mapping algorithm begins with descriptor 1 and maps the lower half of the logical address range by this descriptor. The next descriptor is then used to map the lower half of the remaining address range, and this process continues until only the required 2kb at the top of the addressing range are unmapped in address space FF_{16}. The values in the address space table for the supervisor address spaces are then changed from 0 to FF_{16}.

At this point, descriptors 1 to 12 map the entire addressing range less 2kb in address space FF_{16} (the current address space) and descriptor 0 maps the entire addressing range in address space 0.

Descriptor 0 is then loaded with new data so that it maps the process header for process 0 to address $FFF800_{16}$ in address space FF_{16}. (The physical location of the process header table is set so that process headers are aligned on 2kb physical boundaries. This ensures that they can be mapped using one descriptor).

When the system is operational, only descriptor 0 will be changed when a new process is run. As this was the last descriptor to be loaded, the data for this descriptor remains in the MMU's accumulator. Context switching is therefore easily accomplished by loading the physical base address of the accumulator with the address of the required header and performing a load descriptor operation. This contrasts with the V/68 method of context switching which did not use the memory management units but copied the required header to and from a fixed physical address, and with the method used in the Darkstar Version 7 system which did use the MMU's but needed to load the entire contents of 8 descriptors in order to change context.

7.3 Scheduling

All scheduling is done by the system processor. Scheduling of processes which are running in system mode use the standard non-preemptive scheduling method of the UNIX kernel wherein a process which is taken off the system runq runs until it must wait for an event or until it attempts to return to user mode. When either of these conditions arises a new process is selected and a context switch is effected as described above. The scheduling task has two functions: *a*) to find the highest priority task on the system runq in order to run it (*swtch()*) and *b*) to try to swap in any processes which are swapped out (*sched()*). The first function is always performed when the

scheduling task runs. The second function is only performed when the scheduling task finds itself as the highest priority task on the runq. In the latter situation, it will have been put there because there is some work to be done regarding processes which have been swapped out.

Scheduling of processes which are running in user mode (i.e. on user processors) is done on a preemptive basis and is driven by the clock interrupt. In a multi-processor environment there will generally be more than one current process and the processor table has to be searched to to find these. This is done by the routine ***bdsched()*** which is called at the end of the clock interrupt routine. (All routines which handle the multi-processor environment are in the module "board.c" and are prefixed by "bd").

For each user processor in the system, ***bdsched()*** updates the necessary values for the current process. If the user processor is not currently running a task, the idle time indication in its processor table entry is incremented. If the rescheduling flag is set, the process search routine ***bdswtch()*** is called for each user processor. ***Bdswtch()*** is a general-purpose routine which uses the same algorithm for choosing a process from the runq as the system scheduler ***swtch()***. In addition to this, however, ***bdswtch()*** determines whether the user processor ought to be interrupted or not and its return code indicates this. If the process which is found to be run is the process which is currently running on the user processor, a signal would not normally be sent to the user processor. If that process has a signal pending, however, or if it is being profiled, a signal will be sent to the user processor in order to bring the process into system mode. If the process which is chosen by ***bdswtch()*** is not the current process, but has a signal pending, ***bdswtch()*** removes it from the runq of the user processor and places it on the runq

of the system processor. The search is then repeated to find the next process in the runq with the highest priority.

When a process is run, it is removed from the runq of the user processor by ***bdswtch()*** and its slot number is inserted into the processor table as the next process to be run. If ***bdswtch()*** is called again, before the signal arrives at the user processor, it will remove another process from the runq. The first process to be removed would then effectively be lost, as it will exist on no queues at all and the record of its slot number will have been overwritten. That process will, therefore, never be scheduled again.

To overcome this, ***bdswtch()*** sets a flag in the processor table whenever it manipulates the runq. If ***bdswtch()*** is called again while this flag is set, it will immediately return. The flag is cleared by the VRK just before it begins execution of the new process.

There are two other occasions where ***bdswtch()*** is called to reschedule a user processor. The first is during the processing of an inter-processor signal from a user processor. As this is a normal interrupt, the generic exception handler ***trap()*** is called which takes all of the values on the exception stack frame as its arguments and bases its mode of operation on the type of exception which occurred. Before taking any other action, ***trap()*** checks to see if the exception was an inter-processor signal. If this is the case, the routine ***bdservice()*** is called and when this routine returns, ***trap()*** returns from the exception.

The routine ***bdservice()*** searches the entire processor table for user entries in which the service flag has been set. On finding such an entry, the current process of that processor is placed on the runq of the system processor and the service flag is cleared. Before checking for other service requests or returning to ***trap()***, ***bdservice()*** calls ***bdswtch()***

to find another process to be run on that processor so that it is kept as busy as possible. Due to locking arrangements which are described below, ***bdservice()*** ignores the return value of ***bdswtch()*** and signals the user processor whether there is a new process to be run or not. This does not conflict with the requirements of the system as the user processor, having just requested service, is guaranteed to be idle.

When all processors have been examined, ***bdservice()*** returns to ***trap()*** which returns from the exception. The processes which were found will be scheduled for the system processor in the normal way. It is not an error if, in its search of the processor table, ***bdservice()*** finds no entries in which the service flag has been set for the interrupt may have come from a processor whose request had been dealt with on a previous pass of ***bdservice()***. If a processor is found which has requested service but which is not currently running a process, a message is printed on the system console, but no further action is taken. If, when a process is run by the system, the exception which it experienced is found to have occurred while the user processor was in supervisor mode, this is taken to be an error in the VRK and, in the current implementation, the system halts normal execution of all processes, as it would do if an error were encountered in the kernel.

When a process which is running in system mode attempts to leave the system, it is put on the runq of its processor for normal scheduling. If the processor is idle at the time, however, the process can be run immediately and ***bdswtch()*** is called to schedule it.

7.4 Areas of Difficulty in the Implementation

7.4.1 Processor Allocation

In a multi-processor multi-tasking system, processes are constantly beginning, ending and changing. Some algorithm must be devised for assigning processes to processors in a manner which ensures the maximum overall efficiency of the system. In the current system, the routine *bdfind()* is called to perform this allocation whenever a process creates a new task (*fork()*), overlays another programme in place of the existing task (*exec()*), changes in size, or is swapped in from disk.

In designing the algorithm which *bdfind()* employs to find the appropriate processor for a given task, it was felt that the following points were important:

a) The backplane of the SBC system is used for all off-board accesses. When attempting to access the backplane, a processor may have to retry some bus cycles if another processor is currently using the backplane. This slows down data transfer in all areas. To decrease the probability of retry cycles on the backplane the amount of "traffic" on the backplane should be kept to a minimum and only to data transfers involved in inter-processor signalling between the user and system processors if possible. In the case of shared memory segments, it is not always possible to guarantee that all users of the shared segment reside on the same processor, so the rule is relaxed in this case. If shared text segments resided off-board, however, this would significantly increase the backplane traffic, so a text segment is always provided on the same processor as the process which uses it, even if it involves duplicating the segment on a number of processors.

Although this duplication is possible, it is desirable to avoid it if at all possible, so a process which requires a text segment which is already loaded onto a particular processor should be sited on that processor if possible.

b) If a process is allocated to a processor which does not have enough free memory to accommodate it, swapping must be done in order to process all of the tasks. This too would degrade the overall speed of the system, so processes should be allocated to processors which have sufficient free space to accommodate them if possible.

c) Processors will, in general, have different numbers of tasks allocated to them at any particular time. A processor with a large number of tasks can, in a given period of time, only give a relatively small amount of processing time to each task, provided all tasks are active. Processes should therefore be allocated to the processors with the fewest active tasks. In practice, if a process is on the runq of a *user* processor, it is active, but if a process is not on the runq of a user processor, it is difficult to say whether it is active or not. Processes should, therefore, be allocated to user processors with the fewest active tasks, but, for each processor, this will have to be approximated from the length of the runq and the total number of loaded tasks.

d) Processors should not be allowed to remain idle for long periods of time if the system is to run efficiently. Processes should, therefore, be allocated to processors which have been idle for the longest time.

In the actual implementation, *bdfind()* examines the processor table and the text table. Processors which are not marked as being available are not considered in the search. (The available flag is normally set for each entry in the processor table. If a processor has been dedicated to one real-time task, however, the available flag is cleared so that no further processes will be allocated to that processor. When the real-time process terminates, the available flag is set again and tasks are allocated to that processor as normal).

For a given task, if an available processor has the required text segment and has sufficient free memory to accommodate the process and has been idle for a longer period of time than any other processor with that text segment, that processor is allocated to the task. If such a processor cannot be found, then the existence of the text segment is ignored and the processor which has been idle for the longest period of time is allocated, provided that it contains enough free memory to accommodate the process. If this second criterion also fails to be met, the processor which has the appropriate text segment and the shortest runq is allocated. If all three criteria prove impossible to achieve, the processor which has the required text segment and the fewest tasks is allocated. In this way, processors are kept as busy as possible and the necessity to duplicate text segments is minimised. For the purposes of the algorithm, processes which do not require a shared text segment are conceptually considered as having a text segment on every user processor.

It can be seen from the description of the processor allocation algorithm that, although it attempts to allocate a processor on which the process can be loaded without swapping any other process out, the algorithm always chooses a processor, so it is not guaranteed that the allocated processor contains sufficient free memory for the process. The

calling routine must always attempt to achieve the memory allocation through ***malloc()***, and must take appropriate action if sufficient memory does not exist.

For the current applications, ***bdfind()*** appears to perform adequately and correctly. It should be stressed, however, that ***bdfind()*** is an independent routine and may be replaced by another algorithm without affecting the rest of the kernel or the VRK. This allows the development of better algorithms to achieve the most efficient processor allocation, or the inclusion of algorithms which allocate processors according to a completely different set of criteria (e.g. on a per-user basis).

7.4.2 Race Conditions

In the description so far, it has been implied that signals between the system processor and the user processors are sent whenever they are deemed necessary by each processor. Thus, whenever the system processor wishes to reschedule a user processor, an inter-processor signal is sent to the user processor. In the same way, whenever a user processor requires the system processor to service an exception, a signal is sent to the system processor.

In practice, it is possible for the two conditions to overlap. The user processor may be about to issue a service request to the system processor when the system processor attempts to interrupt the user processor. As all interrupts are masked by the VRK while in the exception handling routine, this interrupt will not be processed until the user processor has interrupted the system processor and is awaiting another signal. Similarly, the system processor may not actually process an interrupt from the user processor until after the user processor has been signalled to begin a new task.

In preparing to issue a signal, however, both processors manipulate data in the processor table. Both processors also modify the contents of this table when processing an inter-processor signal. As both processors run independently, these actions may be interleaved, with the result that either processor (or both) may be acting upon invalid data.

To avoid this situation, a global lock variable (**k_lock**) is maintained in each VRK. The lock is examined and set using the **TAS** (test and set) instruction and is reset using the **CLR** (clear) instruction. The lock may be set either by the kernel or by the VRK but is normally only reset by the VRK, although there is one special case where the kernel resets the lock. The kernel uses two assembly-language routines ***bdlock:*** and ***bdunlock:*** to respectively lock and unlock the VRK. (Although it is actually the data in the processor table which is locked, logically, the VRK is regarded as being the object of the locking procedure). In the VRK itself, the instructions to manipulate the lock are embedded in the code of various routines.

When the kernel attempts to reschedule the user processor as a result of a clock interrupt (i.e. in ***bdsched()***), it attempts to lock the VRK before calling ***bdswtch()***. If this succeeds, the process search is allowed and a rescheduling signal is eventually sent to the user processor. On receipt of this signal, the VRK will clear the lock after it has manipulated the processor table and before it changes context. If the locking attempt by the kernel fails, then the VRK must have locked itself prior to issuing a signal to the system processor. In this case, the process search is not allowed by the kernel, as a new process will be scheduled by ***bdservice()*** after it has processed the signal.

On returning a process to user mode, the kernel may also attempt to reschedule the user processor if its processor table entry indicates that

it is currently idle. As this may be examined just before it is changed by the VRK, however, the kernel must, in this case also, attempt to lock the VRK before issuing the rescheduling signal. If this attempt succeeds, the signal will be sent and the process will be run immediately. If the locking attempt fails, however, then the user processor will have just begun to execute another process so the original process is left on the runq for normal scheduling.

When the VRK encounters an exception condition, it saves the context in the process header. As this occurs when that process was the current process for that user processor, this can be done at any time, for the kernel will not attempt to access that data when the process is not running in system mode. After saving the data, a signal must be sent to the system kernel and, in this case, it must be certain that the kernel will not attempt to reschedule the user processor until the signal has been processed.

The first action of the VRK routine ***signal:*** is, therefore, to attempt to lock itself against a signal from the kernel. If the attempt is successful, the signal is sent and the VRK enters its wait state. The VRK cannot unlock itself until that signal has been processed, however, so it waits in a locked state. The only time the VRK unlocks itself is when it begins to execute a new process. If its runq is empty at this time, however, a scheduling signal will never be sent as ***bdsched()*** will fail to achieve the lock. To prevent deadlock, therefore, ***bdservice()*** must ignore the return value of ***bdswtch()*** and signal the user processor even if there is no process to be run at the time. If the VRK finds that no valid process is indicated when it processes a sheduling signal, it simply unlocks itself and returns to the wait state. (The absence of a

process is indicated by slot number 0 which, as it is the system scheduler, would never be run on a user processor).

It can be argued that it would be simpler for *bdservice()* to unlock the VRK when there is no process to be scheduled, rather than sending a dummy signal to the user processor so that the VRK can unlock itself. If this approach is adopted, however, the operation of *bdservice()* becomes very closely linked to the operation of the VRK and a change in one would probably necessitate a change in the other. Using the current method, the operation of *bdservice()* is independent of the operation of the VRK and the signal which is sent is regarded merely as an indication that the kernel has completed its processing of the service request. The VRK is then able to perform any actions which it deems necessary on receipt of such a signal and will remove the lock from itself only when it has determined that it is time to do so. This same reasoning is used throughout the design of the system and leads to the rule that locks on the VRK must only be removed by the VRK itself. (As mentioned above, there is one exception to this which is discussed below, but in that case, the removal of the lock serves a different function).

In attempting to lock itself prior to issuing a service request, the VRK may find that it fails to acquire the lock. In this case, the kernel is about to issue a rescheduling signal and the service request must not be sent. If the VRK were simply to wait for the scheduling signal, however, the service request would be lost forever for, when execution of the new process begins, the stack will be reset and the exception condition will be lost. When the old process is resumed, it will appear that it has just returned from running in system mode and execution will continue as normal. If the exception was the beginning of a system call,

for example, the system call will appear to have returned without actually having performed any action.

To avoid this, when the VRK cannot lock itself, it sets an unused bit in one of the words in the process header to indicate that the process has been suspended and then enters the wait state. When that process is rescheduled, the ***resume:*** routine first examines and clears this bit. If it finds that the bit had been set then a service request must have been interrupted. In that case, the service request signal is sent immediately and the VRK enters the wait state without attempting to execute the task. The signal can be sent immediately because the lock on the VRK will have been set by the kernel before issuing the scheduling signal. When the service request has been processed by the kernel, the VRK removes the lock in the normal way.

As described above, the kernel locks the VRK before calling ***bdswtch()*** from ***bdsched()***. A signal is only sent to the user processor, however, if it is necessary to do so. In other words, if the new process to be scheduled is the same as the current process, no signal is sent. This implies two things. The first is that, as the VRK unlocks itself after receiving a signal, the VRK will remain locked and no further attempts to reschedule it will be made by the kernel. The second implication is that, during the process search, the VRK may have suspended the current process as a result of encountering the lock condition while attempting to issue a service request.

If the return value of ***bdswtch()*** indicates that no signal is to be sent the kernel unlocks the VRK in order to avoid deadlock. This does not contradict the logic which led to the rule that the VRK alone can determine when its lock is removed, for the assumption was that the setting of the lock achieved some purpose. In this case, the lock

achieved *no* purpose as the data in the processor table remains unchanged. As the setting of the lock was, therefore, unnoticed by the VRK and had no effect on its operation, it *cannot* clear the lock and the lock should be cleared by the kernel.

The situation can arise, however, where the fact that the lock was set *did* have an effect on the operation of the VRK. As mentioned above, if the VRK tried to issue a service request while the lock was set, it would suspend the current process and enter the wait state. In this case, the lock should not be removed as a signal needs to be sent to the VRK so that it in turn will clear the suspended status and issue the service request.

This leads to another race condition, however, for the process may be suspended after its status has been checked by the kernel but before the lock is removed. Conversely, the kernel may have removed the lock and checked the process status after the VRK's unsuccessful attempt to set the lock, but before the process has been marked as being suspended.

To resolve the first condition, the lock is always removed on return from ***bdswtch()*** if it is indicated that no signal is to be sent to the user processor. After the lock is removed, the VRK will not suspend the current process but will issue a service request in the normal way. After removing the lock, however, the kernel checks if the process was suspended before the lock was removed. If this is the case, the lock is replaced and the signal is sent. If the VRK had suspended the process while the lock was in force, it would enter the wait state until a signal was received and would not inspect the state of the lock again. The lock can therefore be replaced immediately after the suspended state is detected and must be replaced before the signal is sent. During the interval that the lock is off, no other kernel activity will access the

VRK either for the only two routines which can do so are ***bdservice()*** and ***bdrun()***. ***Bdservice()*** will not access the VRK as the service request will not have been issued, and ***bdrun()*** will not be active as the kernel will not continue running its current process until after the completion of the clock exception handling of which ***bdsched()*** is a part.

The second condition occurs due to the time lapse between the VRK's checking the state of the lock and its marking the process as being suspended. In practice, therefore, the VRK marks the process as being suspended *before* trying to acquire the lock. If it succeeds in locking itself, the flag is cleared. The kernel may then see the process as having been suspended just before the flag is cleared by the VRK. In this case, however, the kernel is not able to replace the lock and the rescheduling signal signal is therefore not sent.

In summary, therefore, one lock is required on each VRK. This lock will be set by the kernel before attempting to reschedule a user processor. If the scheduling is a result of a clock interrupt but no signal needs to be sent to the user processor, the lock is removed by the kernel. If, however, the user process had been suspended while the process search was in progress, the lock will be replaced and the scheduling signal will be sent.

Locks are removed by the VRK in the routine ***resume:*** provided the process to be resumed is not in the suspended state. If this is so, the lock is not removed and a service request is immediately issued. The VRK sets the lock when it attempts to issue a service request normally. If the lock cannot be set, the current process is marked as having been suspended and the VRK waits until that process is rescheduled before issuing the service request.

7.4.3 Process Transition

In a single-processor system, the transition of user processes between user mode and system mode happens immediately. When an exception occurs due to a user process, it is immediately recognised by the processor and the process (which is already the current process) is continued in system mode. When the exception handling is completed, the return from the exception returns the process to user mode. (This does not happen when the exception causes the process to terminate or when the processor is rescheduled just before the user process is allowed to return from the exception).

In the multi-processor system, however, the transition from user mode to system mode takes three stages. The exception raised by the user process is caught by the user processor but is not dealt with there. The user processor raises an exception on the system processor, upon which the user process is placed on the runq of the system processor (as this processor may be running another process at the time), but, again, the original exception is not handled. In order to deal with the original exception, a third stage is necessary in which the exception is simulated on the system processor. Only at this stage does the kernel handle the original exception. On return from the simulated exception, the user process is replaced on the runq of the user processor and is scheduled normally.

The exception handler for V/68 is the C routine *trap()*. This handles all exceptions raised in the system except interrupts from devices which have their own interrupt routines as a part of the driver. *Trap()* is actually called from a lower-level routine which is called directly from the exception vector. Before *trap()* is called, all internal registers are saved on the system stack above the exception stack frame.

As the arguments to C functions are passed in a stack frame, the arguments to ***trap()*** are the internal registers of the processor and the data in the exception stack frame. On return from ***trap()***, the register values are restored, so ***trap()*** is an unusual C routine in the fact that, if it modifies one of its arguments, the modified argument is copied back. Arguments to ***trap()*** are therefore not passed "by value" as in the case of normal C functions.

To simulate an exception, therefore, ***trap()*** must be called with the appropriate arguments, which are readily available in the process header where they were stored by the VRK (in the same order as as in the kernel's exception stack frame). If ***trap()*** were to be called directly from a C routine, however, the arguments *would* be passed by value and any modifications made by ***trap()*** to the process's registers would be lost.

When a process is first run by the kernel after a service request, the C routine ***bdrun()*** is called. ***Bdrun()*** calls an assembly-language routine ***bdresume:*** which transfers the stored register values from the process header to the system stack. ***Trap()*** is then called from ***bdresume:*** and this will handle the simulated exception in the same way as it would have done on a single-processor system. When the call to ***trap()*** returns, therefore, all the exception processing will have been carried out as normal. ***Bdresume:*** then copies the values from the stack frame back to the process header so that when the process is resumed on the user processor, the VRK will restore the register values returned by ***trap()***.

The function ***bdrun()*** is therefore a simple one as it must only call ***bdresume:*** in order to achieve the complete processing of the exception. On return from ***bdresume:***, ***bdrun()*** places the process on the runq of the user processor as described above. When ***bdrun()*** returns to its calling routine, the system is free to run another user process in system mode.

If the exception caused the termination of the process, ***trap()*** will not return but will call the system scheduler via the ***exit()*** routine. ***Exit()*** carries out the normal functions of releasing the process space, signalling the parent, etc., and will never return. In this case, ***bdrun()*** never continues, so the process is never placed on the runq of the user processor and all traces of it are therefore lost. The complete functionality of the system is therefore retained by this method of handling exceptions.

When a process is scheduled on the system processor for the first time after a service request, ***bdrun()*** must be called. This ensures that the exception is correctly handled by the system and on its return the system scheduler should be called so that other tasks on the system runq will be run. It must be arranged, however, that when ***bdrun()*** returns, the process is left in a suitable state so that, when another service request comes from the same process, ***bdrun()*** is called again. It must also be arranged that, when the scheduler is called after ***bdrun()*** the process is not left in such a state that it will be run again before another service request is issued from that process.

It can, of course, be arranged that ***bdrun()*** never returns but resumes the scheduling task at the end of the function. This would leave the return address on the system stack for that process, however, and after ***bdrun()*** has been repeatedly called, a stack overflow could occur. It must, therefore, be arranged that ***bdrun()*** is called from the same place each time, and that it returns to the same place, so that it may be called again in a subsequent service request.

In switching processes, the UNIX operating system uses two primitives ***save:*** and ***resume:*** which were described in chapter 4. ***Save:*** saves the current context of the process in one of three locations in the

process header and returns immediately with the value 0. *Resume:* recovers the information from the process header which was stored by *save:* and restores the context of the process to the same state as it was when *save:* was called. *Resume:* returns as if it were the call to *save:* returning with the value 1. When it is desired to switch the current process, therefore, a call to *save:* is made and a call to *resume:* is subsequently made with the address of another process as its argument. When the original process is to become the current process again, a call to *resume:* is made with the address of that process as its argument. Execution of the original process then continues from the point after the call to *save:*.

When the system is booted, process initialisation begins with the creation of process 1 which eventually becomes the **init** task. (The process which forks the **init** task eventually becomes the system scheduler: process 0). All other tasks in the UNIX system are descended from the **init** task. The process which eventually becomes the **init** task begins by running in system mode so that it has total control over the initialising of the system. The code which is executed while that process is in system mode is given below:

```
{
        bdproc ();                                  /* boot init */

        while (save (u.u_rsav))              /* save current context */
                bdrun ();                  /* handle service request */

        resume (U, u.u_qsav);                        /* reschedule */
}
```

The variable **u** is of type **(struct user)** which is the structure that resides in the process header. The structure referenced by the variable **u** is always the header of the current process for the system processor.

The elements **u_rsav** and **u_qsav** are arrays within that structure where the context of a process is saved in the event of rescheduling and interrupts respectively. The variable **U** is the array of process headers and is therefore of type **(struct user *)** i.e. a pointer to the structures contained in the process header. The use of the variable ***U*** gives the physical address of the process header table or, in particular, the address of the process header for process 0. (The physical addresses for other process headers is given by the expression (&U[(slot number)])).

When the system is initialised, therefore, the routine ***bdproc()*** is called. This routine handles the physical implications of booting process 1 on a user processor, and is described below. When ***bdproc()*** returns, process 1 is already running on a user processor and will eventually make a system call in order to load "/etc/init". On the system processor, however, ***save:*** is called and will therefore save the context of process 1 at the **while** loop. ***Save:*** returns 0, so the interior of the **while** loop is not entered and ***bdrun()*** is not called. Process 0, the system scheduler is then made to become the current process by calling ***resume:*** and the end of the compound statement shown above is therefore never reached as ***resume:*** never returns.

When the service request is received from process 1, that process will be placed on the system runq (by ***bdservice()***) and will be found by the scheduler which will eventually make it the current process by calling ***resume:***. This will act as if the call to ***save:*** had returned the value 1 and ***bdrun()*** will be called. The service request will be dealt with before ***bdrun()*** returns, as described above. In processing the service request, however, the data in the array **u_rsav** may have been overwritten so, in order to ensure that ***bdrun()*** is called at the next service request, the original contents of **u_rsav** must be restored. This

is the function of the **while** loop. When ***bdrun()*** returns, the ***save:*** routine will be called to evaluate the control expression of the loop. ***Save:*** will again return 0 so ***bdrun()*** will not be called again for this service request but the scheduling process will be run instead. The context of the process will have been saved again at the top of the **while** loop, however, so any subsequent service requests by process 1 will be dealt with by ***bdrun()*** as described.

Process 1 will execute correctly due to the section of code shown above. When a new process is created, process 1 makes a ***fork()*** system call. This will be handled by ***bdrun()*** as described above. The ***fork()*** system call makes an exact copy of the parent process in creating the new process, so the new process will enter user mode by the same route as the parent i.e. by returning to ***bdrun()***. When this invocation of ***bdrun()*** returns to the **while** loop, its context will also be saved at the top of the loop in anticipation of a service request. All new processes in the system will therefore use this route into and out of system mode, so the block of code shown above which was initially tied to process 1 actually forms the core of the multi-processor system and ensures that all processes in the system effect the transition between user mode and system mode in the correct manner.

7.4.4 Process Initialisation

It has been shown above that, once process 1 begins operation on a user processor, the correct operation of the entire system is assured. The task of achieving the first step of this process is done by the routine ***bdproc()*** which is only called once.

The code which will actually cause "/etc/init" to be loaded into memory and executed is held in a known place in the kernel. As all that

is necessary is to make an *exec()* system call having set a few values on the process stack, the initial code is very small. *Bdproc()* chooses the first user processor in the processor table and allocates sufficient memory on that processor to hold the initial code, using the normal routines which manage the MMU table. The initial code only contains a data segment, however, but the values for a nonexistent stack segment are entered by *bdproc()* so that, when process 1 creates new processes, their stack segments will be found in the correct place.

The initial code is then copied to the allocated memory on the user processor and the values which the VRK will retrieve from the process header are all cleared. The processor table entry for the processor is then set up so that it appears to have one process loaded which is the process from slot 1 and this is also made to be the current process for that processor. The scheduling signal is then sent to that processor and *bdproc()* returns.

7.4.5 Shared Memory

Memory resources which are allocated to a single process are easily sited on the same processor. Resources which are shared between processes are more problematical. The case of shared text segments has already been discussed. The elements in the text table now hold multiple references to duplicate text segments, and such segments may be copied directly from processor to processor without using the image on the swap device under certain conditions.

In the current implementation, all other shared memory areas are sited on the system processor. This was done to simplify the programming task, but there is no reason why these segments should not

be sited on arbitrary user processors in a way that would minimise the backplane cycles needed to access them.

7.4.6 Process Isolation

As stated above, it is desirable for a high-priority real-time task to be able to run in isolation on a processor of its own. The operating system guarantees that the execution of any process will not be suspended until it is necessary to do so; isolating a process would add a further safeguard as it will no longer be necessary to interleave the execution of that process with other processes on the same processor. To facilitate this, a new system call, ***suproc()*** has been added to the system. Processes which successfully execute ***suproc()*** become the only active process on a processor. (Such a process is then termed a "***suproc***" from the words "***super process***").

Conceptually, when a process calls ***suproc()***, the kernel finds a suitable processor and removes it from the list of processors which are available for allocation. It then relocates any tasks which are currently running on that processor to other processors and locates the calling process on that processor.

In practice, the processor which is chosen is the processor which the calling process is currently running on, for, even if ***bdfind()*** were called to find a processor, the number of operations which would have to be performed in order to *isolate* the task on that processor would not necessarily be minimised.

When ***suproc()*** is called, therefore, it is desired to move all other processes which are on the same processor as the calling process to other boards. It is difficult to determine, however, whether a process may be moved or not. As the intended suproc is running in system mode

when the call is processed, the user processor on which it resides is likely to be busy running some other process and other processes on that processor may be awaiting some event in system mode. While various flags are set in the kernel to indicate when the location of a process may not be disturbed, it is time-consuming to examine all the possibilities of each process on the the user processor and, if it is the case that one or more processes could not be moved at that time, there would be nothing that could be done until that condition were removed (which may be a long time). (Attempting to site the intended suproc on another board does not necessarily solve the problem, as all boards may contain processes which are locked in core).

The solution which is employed is not to attempt to move any processes from a board when *suproc()* is called. The correct operation of a suproc would only be disturbed if some other process returns to user mode while the suproc is executing. At the time when *suproc()* is called, the intended suproc is in system mode and is therefore unaffected by user mode processes on its processor. After the completion of the call, the suproc will return to user mode by being scheduled as normal. After this, it is essential that no other process is allowed to execute in user mode on that processor. This is easy to do as, when a process is about to resume execution in user mode, it cannot be locked in core. Such a process can therefore be moved to another processor at that point. Processes on the suproc's processor which never run in user mode during the life of the suproc will never be removed from the processor but, as such processes do not use the user processor, they do not affect the operation of the suproc. The detection of the condition where an ordinary process which resides on a processor where a suproc is running

attempts to return to user mode is, therefore, the central operation in supporting the execution of suprocs.

When ***suproc()*** is called, provided there are sufficient available processors in the system for one to be taken off the available list, a flag is set in the process table which marks the calling process as being a suproc and its processor is marked as unavailable for allocation. The suproc is then placed on the runq of the processor to be scheduled as normal.

Any new processes which are created will not be allocated to that processor as ***bdfind()*** will not attempt to allocate a process to an unavailable processor. Processes which are already resident on the processor must all be inactive at the time when the suproc returns to user mode. When ***bdswtch()*** examines the runq of the processor to choose the next process to be executed, it checks the status of the process and the processor. A process which is not marked as a suproc but which is about to be scheduled on a board which is not available is swapped out. When it is swapped in again, it will be allocated to an available processor. Using this method, the suproc is allowed to enter the kernel in the normal way and, when re-entering user mode, will be resumed on the same processor.

A race condition occurs when a process calls ***suproc()*** before the completion of a ***suproc()*** call by another process on the same board. In this case, one of the processes finds its processor marked as unavailable when it begins execution in system mode. This process is swapped out and later swapped in to another board before its execution is continued. Again, the same race may occur on the new processor so the process will be repeated. In the end, either a suitable processor will be found or the

number of available boards will have fallen to the minimum number, in which case the call to *suproc()* will fail.

The kernel swapping routine *sched()* will not attempt to swap out a process which has been marked as a suproc.

7.5 Changes in the System Superstructure

The multi-processor operating system was designed to provide the same user interface as normal single-processor versions of the system. The vast majority of of the V/68 utility programmes run on the new system without the need for modification although some had to be recompiled to use the modified structures contained in the system header files. Those programmes which require modification to the actual programme code are the same category of programmes which required modification when the new memory management system was introduced on the Darkstar Verion 7 system i.e. programmes which explicitly reference physical memory.

The utility '*ps*' is a good example of this. *Ps* searches the process table, the process header table, the MMU table, the text table and the stack segment of processes in order to provide the user with a status report for each process of interest. As those system tables have changed in format (i.e. they are now all arranged by slot number) *ps* had to be changed so that it correctly referenced the new table layout. In the case of *ps*, though, the process table now contains one additional entry which would also be of interest to the user - the board number which the process is running on. *Ps* was, therefore, also modified to output this information.

The execution of real-time processes is aided by the addition of the system call *suproc()*. This call is made when a process is required to be

run on a dedicated processor in order to avoid time-sharing with other processes. If *suproc()* is successful, it will return a small positive integer to the calling routine to indicate that the process is now located on a processor of its own. (In this implementation, the number returned is, in fact, the processor number of the allocated processor. This number may be used by hardware-dependent routines). The calling process then becomes a suproc. *Suproc()* will fail and return the value -1 if the effective user identification of the process is not that of the super-user or if insufficient processors are available in the system to allow one to be dedicated to the process. (In the current implementation, the system ensures that at least one user processor is always available to run ordinary, non-dedicated tasks e.g. *'init'*).

At present, a process which has become a suproc retains that status until it exits. Suproc status is retained across *exec()* calls but is lost across *fork()* calls.

7.6 Summary

This chapter has described the actions and interactions of the major routines which provide the functionality of the multi-processor operating system.

The basic ideas behind the design of the system were, in general, fairly easy to translate into the design of the individual routines, although there were some areas (e.g. process transition) where a lot of the auxilliary code had to written while ideas for the main function were being considered and rejected.

Debugging the system also presented new difficulties, as it was often unclear whether the source of an error was in the VRK or in the kernel, as neither could be tested in isolation. Both areas had to be

examined, therefore, and there were numerous occasions where what seemed to be obviously a fault in one area turned out to be a fault in the other.

The processor allocation algorithm exists, in its present form, as the basis for an evaluation of various allocation methods. While it works adequately in providing a near-equal distribution of processes to processors, it is easy to construct scenarios in which its method of allocation would be less than ideal. As all processor allocation is performed by the one routine, it would be easy, in the future, to substitute an improved routine in its place. Some areas of improvement may, however, require that more information be kept in the processor table and this would, in turn, require that a number of modules would need to be re-compiled.

The area which provided most difficulty was the design of the VRK locking mechanism. Not only was it difficult to identify the actual race conditions, but, in doing this, it was often attempted to correct "race conditions" which did not actually exist. It would be difficult to prove that all race conditions have now been identified and corrected, but the system has been operational in a multi-user, multi-processor mode for a number of months at the time of writing and no new race conditions have been detected so it can be assumed that the present locking arrangement is adequate.

Chapter 8

General Assessment

The aim of this project has been to develop an environment wherein real-time processes could be efficiently developed, tested and executed. By taking advantage of the decreasing cost of microprocessors and random access memory chips, a multi-processor solution to the problem has been designed and implemented, using a popular, readily-available, commercial operating system as its basis.

The final system provides an appreciable increase in speed over the single-processor version of the operating system, although the lack of any accredited benchmarks for comparing multi-processor, multi-tasking systems against their single-processor counterparts has precluded exhaustive comparative testing. The problem was compounded by the inability to predetermine what the operating system or any user process would be doing at any particular time.

In the end, a modified version of the "Sieve of Eratosthenes" was used to provide some measure of comparison between various environments. As the same object code was to be used, the programme was written and compiled without any attempt at optimisation for, the longer it took to execute, the less would be the effects of timing errors due to the granularity of the timing operations which are available in the UNIX system.

The UNIX system call which was used to provide an indication of both the elapsed time and the time actually spent executing the process (processor time) was *times()* which is accurate to 0.02 seconds. Unfortunately, the UNIX system does not provide any more accurate means of timing than this. As the test programme was intended to run for some

10 seconds or more, however, it was felt that *times()* would provide an adequate indication for comparative purposes.

In order to simulate a busy system in a controlled manner, the test process optionally generated 3 child processes which performed dummy operations and accessed the system (via *alarm()* and *signal()*) at random times while the test was proceeding. The test routine itself did not perform any system calls as it was intended to simulate a real-time process which would communicate with its support tasks via shared memory. The source code of the test programme is given in Appendix 3.

The test was run in single-user mode under 5 conditions:

a) On the single-processor system as the only active task.

b) On the single-processor system along with 3 dummy tasks.

c) On the multi-processor system as the only active task.

d) On the multi-processor system along with 3 dummy tasks.

e) As a suproc on the multi-processor system along with 3 dummy tasks.

(The multi-processor system which was used in these tests had one system processor and two user processors). The results of these tests are given in Fig. 8.1.

Tests c) and e) appear to give 100% efficiency but this is somewhat misleading. Using this implementation of the system, the tests actually measure the ratio of the time a task is handled by the user processor to the total elapsed time. This is near to the efficiency ratio, but there is some disparity, as the user processor uses some of the time to drive the memory management system.

Nevertheless, less memory management is required in test c) than in test a) as the MMU descriptors are not disabled every time the system scheduler is resumed. The potential for improvement increases in test e)

for, if the inverse mapping arrangement described in chapter 7 is implemented, no memory management at all will be done by the user processor.

The advantages of the multi-processor system as implemented in this project over its single-processor counterparts are that the multi-processor system allows user tasks to execute concurrently and also allows user tasks to run concurrently with system activities. The result is an increased overall throughput of user tasks in a given period of time, and the opportunity for the correct execution of real-time tasks. The multi-processor system also has the advantage of being able to be expanded to provide increased processing power. Single-processor systems can only be expanded to provide increased storage capacity.

The disadvantages of the multi-processor system as compared to a single-processor system are that the multi-processor system is more complex, more costly, and that calls to the system under a multi-processor system take longer to reach the system than they would take under a single processor system.

While it is true that the overall complexity of the system is increased by additional processors, the division of tasks among those processors means that the operation of each module in the system actually becomes simpler.

The cost of microprocessors is, as has been shown, much less than the cost of certain other elements of computing hardware and it is possible to significantly increase the number of microprocessors in a computing system (and thereby increase the processing power of the system) for a small fractional increase in the cost of the overall system.

The time taken between an exception occurring in a user process and its appearance on the system processor as an invocation of the exception handling routine is certainly greater than it would be under a single-processor system. In general, however, the user process would have been executing at times when it would not have been able to have done so, had it been running under a single processor system, due to the necessity of having to share processing time with system activities.

Although it is true, therefore, that the time taken from the beginning of the exception to the beginning of its processing on the system processor would be increased, the time taken from the beginning of the task to the return from the exception would, in general, be decreased.

8.1 Opportunities for Further Work

The operating system in its current form was written to achieve the quickest implementation of the essential elements of the specification. One result of this is that many sections of the design and of the code have not been optimised.

For example, the UNIX system differentiates between process information which can be swapped out with the process and information which is needed about the process even when it has been swapped out. The former category of information formed the process table, while the latter category resided in the process header. The two sets of information were cross-referenced.

In the implementation of the multi-processor system, it is desirable to have the information of the process header in a fixed location and one which is easily accessible to the system processor. Because of this, the process headers are stored in a table on the system processor. To reduce the number of changes to the code, however, the process table and the

process header table are still maintained as two separate tables, although it is now possible to combine both categories of information into one table.

There are, in the present implementation, a number of areas like this which, if they were rationalised, would speed up the operation of the system as a whole and reduce the complexity of the code. The process of rationalisation would require extensive changes to be made throughout the kernel, however.

The above example shows that the present implementation (and future enhancements of this implementation) are constrained by the present structure of the UNIX kernel which was designed for a different class of applications. It may, therefore, be advantageous in the future, to rewrite the entire kernel so that it operated in a manner which is more conducive to the interaction with the multi-processor features which have been introduced.

Another limiting factor in the efficiency of the present system is the memory management system. The current system is based on the skeletal driver which was delivered as a part of the V/68 system and therefore exhibits the same strengths and weaknesses, although its shortcomings do not affect system performance as much as on the single-processor system. As only one address space number is used for all user processes, all MMU descriptors must be disabled when a new process is scheduled, and the processing of page faults must begin again from scratch.

In the multi-processor system, the effects of this are alleviated by the fact that processes are only rescheduled when absolutely necessary and need not be preempted in order to run the system scheduling process. Nevertheless, a large proportion of processing time is still spent in

driving the memory management units, especially in the period shortly after a process has begun execution.

If the MMU table were structured so that different address space numbers were allocated to different processes, the situation would improve, although this would probably have to be incorporated alongside a scheme where descriptors are allowed to define segments of greater than one click if a substantial improvement is sought. The scope for improvement in the multi-processor system is even greater than in the single-processor system as there will be fewer processes being serviced by a particular MMU. For each process, therefore, a larger number of descriptors will be allowed to remain enabled while the process is not running, with the result that, when the process is resumed, a larger proportion of the previous mapping will have been retained. (In the present system, none of the previous mapping is retained).

In the single-processor system, the use of multiple address space numbers would impose a (fairly large) limit on the maximum number of processes which the system allows at any particular time, or would require a complex hybrid system which allowed some sharing of address space numbers. Again, on the multi-processor system, this problem will be alleviated as the limit will only be imposed on the number of processes per processor.

The current implementation of the multi-processor system uses one system processor and many (theoretically, up to 60) user processors. The system processor performs all I/O operations. This has two implications.

The first implication is that all system activities must be run on the same processor, which allows no concurrency of system operations. For example, the system activities concerned with I/O transactions, could be performed in parallel with the general housekeeping activities which

are initiated by a clock interrupt. Such parallelism would speed up the overall performance of the system.

Another method of achieving concurrency of system operations would be to allocate system processors to system tasks in the same way as user processors are allocated to user tasks. This would allow the system to have more than one current process. This method would, however, involve considerable re-programming of the system, as the operation of many areas of the kernel implicitly relies on the fact that only one current task will exist in system mode.

As the system processors perform more complex tasks than the user processors, a considerably more complex communication and locking system will have to be devised for the system processors, and an evaluation would have to be done to determine whether the increased complexity will result in a sufficient enhancement of the operation of the system to be worthwhile.

The second implication of the present structure of the system is that processor boards which are used as user processors have a lot of operational potential which cannot be used while operating in that mode. As the system processor cannot access the I/O page of the user processors, and as the user processors perform no I/O, all the I/O functions of the user processors are idle.

Fortunately, the actual I/O chips may be absent from the processor boards while the boards are operational, so the cost of populating a user processor board is less than that of populating a system processor board, and the idle hardware does not have to be paid for. If it were required to extend the system hardware, however, (to have both a parallel interface and a Winchester disk controller interface, for example), a separate board would have to be added to the system to provide this

extra functionality, although the required physical capacity would already exist on a user processor.

If the user processors were allowed to access their I/O pages, it would defeat the intended mode of operation of the system as a whole, so the problem is essentially one which is connected with the hardware design.

The multi-processor system, as currently implemented, offers the possibility of real-time execution of user tasks. These tasks must, at present, be simulations, however, as no facility exists for the task to interface to external hardware, in order to perform control functions.

The UNIX system provides a memory driver which allows processes to access physical memory outside their own address space, so it would be possible for a task, using this driver, to transfer data to and from external hardware registers. It should be noted, however, that, compared to normal read and write operations, the transfer of data using the memory driver is slow and cumbersome.

The major obstacle to real-time control is, however, the inability of the present implementation to provide an interrupt facility between external hardware and the user task. In order to achieve this mode of operation, which would be the final step in creating an operating system which would accomodate a full range of real-time processing applications, a mechanism must be devised whereby a hardware interrupt is converted to a UNIX signal which would then be sent to the control task.

Solutions to both these problems, involving the generalisation of the shared memory operations and the ***signal()*** system call, have been discussed in chapter 6 but have not been included in the present implementation. As the design of digital control systems is one of the major interests of the author, it is felt that the addition of this mode

of operation to the current implementation would be the most fruitful avenue of further research arising from this project.

Where real-time processing is not required, there are still areas where the multi-processor aspects of this project may be developed. For example, a UNIX system may be developed where all user terminals are, in fact, workstations, each with its own processor and local memory. The major modification to the existing system would be the simplification of the routine ***bdfind()*** to allocate processes to the requesting workstation. Further improvements may then be considered such as moving the terminal driver into the VRK in order to relieve the system processor(s) of the job of character processing.

One potential area of difficulty which has not surfaced in the current implementation is the problem of frequently occurring interrupts. The system clock interrupts at priority 6 every 0.02 seconds. During the time it takes to process this interrupt, all other sources of interrupts in the system are effectively disabled, as the clock is the highest priority interrupting device. This implies that the clock interrupt handler must be short or the possibility of losing some of the lower level interrupts arises (e.g. a serial interface adapter may encounter a data overrun if its interrupt request is not serviced within the time taken to receive a character).

Fortunately, the normal operation of the clock interrupt handler is not very time-consuming. Every fiftieth clock interrupt is, however, very lengthy, as the system processor then attempts to inspect the state of all the user processors and reschedule all of them. As more user processors are added to the system, this activity, which occurs with all lower interrupt levels masked, may well take too long to complete if there are high-speed I/O devices in the system.

A solution would be to have the clock interrupt prime some other interrupt source which interrupts the processor at level 1. The clock interrupt handler would never attempt the rescheduling activity, as this would now be done by the level 1 handler. This would allow all other interrupts to be active while the rescheduling operation is in progress.

8.2 Conclusions

A number of benefits are gained by having an operating system under which tasks can be run in real-time. Using such an operating system allows tasks to be developed and tested in the same environment in which they will eventually be run. The environment created by a multi-tasking operating system also facilitates the transfer of data between projects, some of which may be concerned with real-time software applications, and others, whose goals may be more general.

This thesis has demonstrated that, in spite of many conflicting criteria, it is possible to design an operating system which combines real-time aspects with a broad, accessible user interface, such as that which is normally present in popular operating systems.

The results have shown that the entire system benefits by the inclusion of real-time aspects into an operating system. All processes (including system processes) execute with fewer interruptions and a greater throughput of tasks is realised along with a decrease in the probability of system deadlock.

The system which has been developed may be easily modified to suit other multi-processor applications which do not require real-time processing.

References and Relevant Publications

1. Ayache, J.M., Courtiat, J.P., Diaz, M. — *REBUS, A Fault-Tolerant Distributed System for Industrial Real-Time Control*

   IEEE Transactions on Computers, Vol. C-31, No. 7, July 1982

2. Bach, M.J., Buroff, S.J. — *Multiprocessor UNIX Systems*

   AT&T Bell Labs Technical Journal, Vol. 63, No. 8, October 1984

3. Barron, D. — *Computer Operating Systems for micros, minis and mainframes*

   Chapman and Hall, London, 1984

4. Barskiy, A.B., Savvin, I.M. — *An Operating System for Dynamic Parallel Allocations in a Control Computer System*

   Cybernetics, Vol. 11, No. 3, May-June 1973

5. Bayer, D.L., Lycklama, H. — *The MERT Operating System*

   Bell System Technical Journal, Vol. 53, No. 6, July/August 1978

6. Bodenstab, D.E., *UNIX Operating System*
Houghton, T.F., *Porting Experiences*
Kelleman, K.A.,
Ronkin, G.,
Schan, E.P.
AT&T Bell Labs Technical Journal, Vol. 63, No. 8, October 1984

7. Bourne, S.R. *The UNIX System*
Addison Wesley Publishing Co., London, 1982

8. Brookes, G.R., *A Practical Course on Operating Systems*
Theaker, C.J.
Macmillan, London, 1983

9. Capo, J.O., *A Real Time BASIC Control System*
Foster, H.R.
ISA Transactions, Vol. 12, No. 4, 1973

10. Conrad, V., *Parallel Solutions of Load Flow Problems*
Wallach, Y.
Archiv für Elektrotechnik, No. 57, No. 6, March 1976

11. Dijkstra, E.W. *Co-operating Sequential Processes*
Programming Languages, F. Genuys(Ed.), Academic Press, New York, 1968

12. Dolotta, T.A., *The Programmer's Workbench*
Haight, R.C.,
Mashey, J.R.
Bell System Technical Journal, Vol. 53, No. 6, July/August 1978.

13. Dorman, R.G., *A Real Time Operating System*
Kopp, R.S., *for Manned Spaceflight*
Weiler, P.W.
IEEE Transactions on Computers, Vol. C-19, No. 5, 1970

14. Driscoll, C.G., *A Processor Allocation Method for*
Mullery, A.P. *Time Sharing*
Communications of the ACM, Vol 13, No. 1, 1970

15. Duncan, T., *Software Structure of No. 5 ESS - A Distributed*
Huen, W.H. *Telephone Switching System*
IEEE Transactions on Communications, Vol. COM-30, No. 6, June 86

16. Dy Liacco, T.E. *Real-time computer control of*
power systems
Proceedings of the IEEE, No. 62, 1974

17. Enslow, P.H., *Parallel Control of Distributed Systems*
Saponas, T.G.
Parallel Processing Systems, CUP, 1982

18. Giorcelli, S. *Sistema di controllo per elaborati di processi operanti in parallelo*
CSELT Rapporti tecnici, No. 1, April 1975

19. Gladman, J.C., Kingsbury, M.A. *The M2140 Computer: A Multiprocessor System for Real-Time Control*
Journal of Science & Technology, Vol. 36, No. 3, 1969

20. Hansen, P.B. *The Nucleus of a Multiprogramming System*
Communications of the ACM, Vol. 13, No. 4, 1970

21. Johnson, H.H., Shearer, B.R. *Ein Betriebssystem für verschiedenartige, parallel anfallende Programme*
Regelungstechnische Praxis und Prozeß-Rechentechnik, April 1972

22. Kernighan, B.W., Ritchie, D.M. *The C Programming Language*
Prentice Hall Inc., Englewood Cliffs, NJ, 1978

23. Krella, F.W. *Language Definition EUROCORAL TR4*
CEE Brétigny, November 1970

24. McDonald, A.R. *Minicomputers...their places in the sun*
Data Management, Vol. 14, No. 2, February 1976

25. McDowell, R.B. *The APL technical approach to real-time, interactive, multiple-computer systems*
Simulation, Vol. 17, No. 1, July 1971

26. Menga, G., Zoppoli, R. *MODIAC: il sistema distribuito del Consiglio Nazionale delle Ricerche per il controllo dei processi industriali* **Automazione e Strumentazione**, May 1982

27. Motorola *M6800 Microprocessor Programming Manual* **Motorola**

28. Motorola *MC68000 16-bit Microprocessor User's Manual* **Motorola**

29. Motorola *MC68010 User's Manual* **Motorola**

30. Motorola *MC68020 User's Manual* **Motorola**

31. Naeini, R. *A few statement types that adapt the C language to parallel processing* **Electronics**, June 28, 1984

32. Nielsen, N.R. *Controlling a real-time parallel processing computer* **Simulation**, Vol. 17, No. 3 September 1971

33. Norrie, C. *Supercomputers for Superproblems: An Architectural Introduction* **Computer**, March 1984

34. van Oost, E.M.J.C. *Multi-Processor System Description and Simulation Using Structured Multi-Programming Languages* **Computer Architecture News**, Vol 9, No. 2, April 1981

35. Pyle, I.C. *The ADA Programming Language* **Prentice Hall International**, London, 1981

36. Richards, M., Whitby-Strevens, C. *BCPL the language and its compiler* **Cambridge University Press**, Cambridge, 1980

37. Ritchie, D.M. *UNIX: A Retrospective* Bell System Technical Journal, Vol. 53, No. 6, July/August 1978

38. Schönherr, H.J. *Teil 5: VM/370 - Systemsteuerprogramm für den Parallelbetrieb virtueller Maschinen* **IBM Nachrichten**, Vol. 22, No. 212, October 1972

39. Simon, K.D. *PEST Prozeßorientietes Echtzeitsteuerprogrammsystem für K1520* **rechentechnik/datenverarbeitung**, March 1980

40. Slade, A.J. *Distributed control system design* **Department of Computing Internal Report**, University of Durham, 1982

41. Slade, A.J. *Industrial Distributed Control* **IUCC Bulletin**, Vol 5, No. 1, Spring 1983

42. Stone, H.S. *Parallel Computers*

Introduction to Computer Architecture, SRA, Palo Alto, CA, 1975

43. Thurber, K.J. *Associative and Parallel Processors*

Wald, L.D.

Computing Surveys, Vol. 7, No. 4, December 1975

44. Timmesfeld, K.H. *Pearl - a proposal for a process and experiment automation real-time language*

G. f. K., Karlsruhe, PDV-Bericht KFK-PDV1, April 1973

45. Whitworth, P.F. *A Multi-Family Multi-Processor Education and Development System*

Ph.D. Thesis, University of Bath, 1983

46. Zaks, R. *The CP/M Handbook*

Sybex, Berkeley, CA, 1980

47. Zaks, R. *Programming the Z80*

Sybex, Berkeley, CA, 1982

48. Zima, H. *PROGRESS - eine Programmiersprache für Realzeitsysteme*

Angewandte Informatik, Vol. 16, No. 8, August 1974

Acknowledgements

I would like to thank the following people who provided me with assistance and encouragement during my period of research and during the time of writing this thesis:

Dr. P.F. Whitworth, for his supervision of the project and his assistance with the solution of numerous problems.

Mr. A.R. Daniels, for his help and support of the project and for effectively taking over the supervision of the final stages.

Prof. J.F. Eastham and Prof. T.E. Rozzi, for the use of the School of Electrical Engineering and its resources.

Mr. V.S. Gott and his staff, for their help with hardware faults and anomalies.

Mr. P. Carr, Dr. R. Yardley and the staff of the Computer Unit, for their assistance and the use of their facilities.

Dr. A. Brown, Mr. L. Dale and my fellow postgraduate students, for their time and assistance which was always freely given.

The staff of the Electrical Engineering School Office, for always being on hand to sort out the non-technical difficulties.

Mr. C. Wildridge, for procuring a mass of literary material.

Clare, for her assistance with the typing and compilation.

Mary, Mary, Clare and Chris, who proof-read this thesis, even though they had no interest in computing or engineering.

All my friends and family, who supported my work and gave encouragement throughout.

Finally, I would like to specially thank Penny for her friendship and encouragement in the final stages of my research and for continually motivating me until this thesis had finally been completed.

Appendix 1

A Semi-formal Specification of the C Language[22]

1. Expressions

The basic expressions are:

expression:
primary
* *expression*
& *expression*
\- *expression*
! *expression*
~ *expression*
++ *lvalue*
-- *lvalue*
lvalue ++
lvalue --
sizeof *expression*
(*type name*) *expression*
expression binop expression
expression ? *expression* : *expression*
lvalue asgnop expression
expression , *expression*

primary:
identifier
constant
string
(*expression*)
primary (*expression-list*$_{opt}$)
primary [*expression*]
lvalue . *identifier*
primary -> *identifier*

lvalue:
identifier
primary [*expression*]
lvalue . *identifier*
primary -> *identifier*
* *expression*
(*expression*)

The primary expression operators

() [] . ->

have highest priority and group left-to-right. The unary operators

* & - ! ~ ++ -- **sizeof** (*typename*)

have priority below the primary operators but higher than any binary operator, and group right-to-left.

Binary operators and the conditional operator all group left-to-right and have priority decreasing as indicated:

binop:

```
*    /    %
+    -
>>   <<
<    >    <=   >=
==   !=
&
^
|
&&
||
?:
```

Assignment operators all have the same priority, and all group right-to-left.

asgnop:

```
=    +=   -=   *=   /=   %=   >>=  <<=  &=   ^=   |=
```

The comma operator has the lowest priority, and groups left-to-right.

2. Declarations

declaration:
decl-specifiers *init-declarator-list*$_{opt}$;

decl-specifiers:
type-specifier *decl-specifiers*$_{opt}$
sc-specifier *decl-specifiers*$_{opt}$

sc-specifier:
auto
static
extern
register
typedef

type-specifier:
char
short
int
long
unsigned
float
double
void
struct-or-union-specifier
enum-specifier
typedef-name

init-declarator-list:
    init-declarator
    init-declarator , *init-declarator-list*

init-declarator:
    declarator initializer$_{opt}$

declarator:
    identifier
    (*declarator*)
    * *declarator*
    declarator ()
    declarator [*constant-expression*$_{opt}$]

struct-or-union-specifier:
    struct { *struct-decl-list* }
    struct *identifier* { *struct-decl-list* }
    struct *identifier*
    union { *struct-decl-list* }
    union *identifier* { *struct-decl-list* }
    union *identifier*

struct-decl-list:
    struct-declaration
    struct declaration struct-decl-list

struct-declaration:
    type-specifier struct-declarator-list ;

struct-declarator-list:
    struct-declarator
    struct-declarator , *struct-declarator-list*

struct declarator:
    declarator
    declarator : *constant-expression*
    : *constant-expression*

initializer:
    = *expression*
    = { *initializer-list* }
    = { *initializer-list* , }

initializer-list:
    expression
    initializer-list , *initializer-list*
    { *initializer-list* }

abstract-declarator:
    empty
    (*abstract-declarator*)
    * *abstract-declarator*
    abstract-declarator ()
    abstract-declarator [*constant-expression*$_{opt}$]

type-name:
    type-specifier abstract-declarator

enum-specifier:
    enum { *enum-list* }
    enum *identifier* { *enum-list* }
    enum *identifier*

enum-list:
    enumerator
    enum-list , *enumerator*

enumerator:
    identifier
    identifier , *constant-expression*

typedef-name:
    identifier

3. Statements

compound-statement:
    { *declaration-list*$_{opt}$ *statement-list*$_{opt}$ }

declaration-list:
    declaration
    declaration declaration-list

statement-list
    statement
    statement-list

statement:
    compound-statement
    expression ;
    if (*expression*) *statement*
    if (*expression*) *statement* **else** *statement*
    while (*expression*) *statement*
    do *statement* **while** (*expression*) ;
    for (*expression-1*$_{opt}$; *expression-2*$_{opt}$; *expression-3*$_{opt}$) *statement*
    switch (*expression*) *statement*
    case *constant-expression* : *statement*
    default : *statement*
    break ;
    continue ;
    return ;
    return *expression* ;
    goto *identifier* ;
    identifier : *statement*
    ;

4. External definitions

program:
    external-definition
    external-definition program

external-definition:
    function-definition
    data-definition

function-definition:
    type-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
    declarator (*parameter-list*$_{opt}$)

parameter-list:
    identifier
    identifier , *parameter-list*

function-body:
    type-decl-list function-statement

function-statement:
    { *declaration-list*$_{opt}$ *statement-list* }

data-definition:
    extern$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;
    static$_{opt}$ *type-specifier*$_{opt}$ *init-declarator-list*$_{opt}$;

5. Preprocessor

#define *identifier token-string*
#define *identifier*(*identifier* , ... , *identifier*) *token-string*
#undef *identifier*
#include "*filename*"
#include <*filename*>
#if *constant-expression*
#ifdef *identifier*
#ifndef *identifier*
#else
#endif
#line *identifier*

Appendix 2

Manual Page for the SUPROC() System Call

SUPROC(2) (SBC only) SUPROC(2)

NAME

suproc - elevate process status

SYNOPSIS

int suproc ()

DESCRIPTION

The calling process becomes a suproc - the only process running on a particular processor in a multi-processor system.

Suproc will fail if one or more of the following are true:

The effective user id of the process is not super-user. [EPERM]

There are too few available processors in the system for the call to be executed without seriously damaging the performance of the system. [EINVAL].

Suproc remains active across *exec* system calls. *Suproc* does not remain active across *fork* system calls.

RETURN VALUE

Upon successful completion, a positive integer, namely the processor number, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

WARNING

There is no way of removing the effect of *suproc* other than by *exit*.

SEE ALSO

exec(2), exit(2), fork(2).

Appendix 3

The "Sieve of Eratosthenes" Test Programme

Listed below is a version of the "Sieve of Eratosthenes" prime number search algorithm. Due to the enforced granularity of the timing, this version was written to be slow. This programme was run under the conditions described in Chapter 8 to provide a comparison of the single-processor and multi-processor systems.

```
/*
 * File: "sieve.c"
 * Usage: sieve [n]
 * Arguments:
 *    n omitted:      NCHILD children
 *    n < 0:          Debug mode - no children, output primes
 *    n == 0:         no children
 *    n > 0:          NCHILD children search as a suproc
 * Output: Ratio of processor time to elapsed time
 */
#include  <signal.h>
#include  <setjmp.h>
#include  <sys/param.h>
#include  <sys/types.h>
#include  <sys/times.h>

#define   NCHILD    3                  /* Number of children spawned */
#define   NSEEK     0x10000              /* Number of primes sought */
#define   INTTIME   6          /* maximum time between system calls */
#define   LLEN      10                          /* Output line length */
#define   ISPRIME   1
#define   NOTPRIME  0

char list[NSEEK];                                 /* Table of Primes */
jmp_buf freewheel;
int sigcatch(), long times();

sieve ()                                          /* find prime numbers */
 {
     unsigned i, j, nprime;

     for (i = 0; i < NSEEK; i++)
          list[i] = ISPRIME;
     list[0] = NOTPRIME, nprime = 1;
     for (i = 2; i < NSEEK; i++)
      {
          if (list[i] == NOTPRIME)
               continue;
          nprime++;
          for (j = i + i; j < NSEEK; j += i)
               list[j] = NOTPRIME;
      }
     return (nprime);
 }
```

```
main (argc, argv)
char **argv;
 {
     int i, nprime;
     long start, duration;
     char wflg = 0, cflg = 1, sflg = 0;    /* output, child & suproc */
     int ctab[NCHILD];                          /* pid's of children */
     struct tms cputime;

     if (argc > 1)
          if ((i = strol (argv[1], 0, 0)) < 0)
               wflg++, cflg--;  /* debug mode - output, no children */
          else if (i == 0)
               cflg--;              /* unloaded system - no children */
          else sflg++;                          /* suproc + children */
     if (sflg)
          if (suproc () < 0)
           {
               printf ("sieve: Can't become suproc\n");
               exit (1);
           }
     if (cflg)                                          /* load system */
          for (i = 0; i < NCHILD; i++)
               if ((ctab[i] = fork ()) == 0)
                {
                    child ();
                    exit (0);
                }
     start = times (&cputime);
     nprime = sieve ();
     duration = times (&cputime) - start;          /* time operation */
     if (cflg)                                    /* terminate children */
          for (i = 0; i < NCHILD; i++)
               kill (ctab[i], SIGKILL);
     if (wflg)
          prprime (nprime, duration);           /* list found primes */
     else average (nprime, duration, cputime.tms_utime);
     while (wait ((int *)0) >= 0)                    /* bury children */
          ;
     exit (0);
 }

child ()
 {
     int x;

     srand (getpid () ^ time ((long *)0));
     if (setjmp (freewheel) == 0)
          sigcatch ();
     for (;;)
          x = (x << 1) + 1;
 }
```

```
prprime (nps, t)
 {
     register i, n;

     for (i = 0, n = 0; i < NSEEK; i++)
      {
          if (list[i] == NOTPRIME)
               continue;
          printf ("%d", i);
          if (++n < LLEN)
           {
               printf ("\t");
               continue;
           }
          n = 0;
          printf ("\n");
      }
     if (n)
          printf ("\n");
     printf ("Elapsed time (debug mode, %d primes) = %d\n",nps, t/HZ);
 }

average (nps, t, ut)        /* t = elapsed time, ut = processor time */
 {
     float et, pt;

     pt = (float)ut;
     et = (float)t;
     pt /= et;
     printf ("Efficiency ratio (%d primes) = %g\n", nps, pt);
 }

sigcatch ()
 {
     signal (SIGALRM, sigcatch);
     alarm (rand () % NCHILD + 1);
 }
```

ILLUSTRATIONS

Figure 2.1

M68000 Memory Organisation[28]

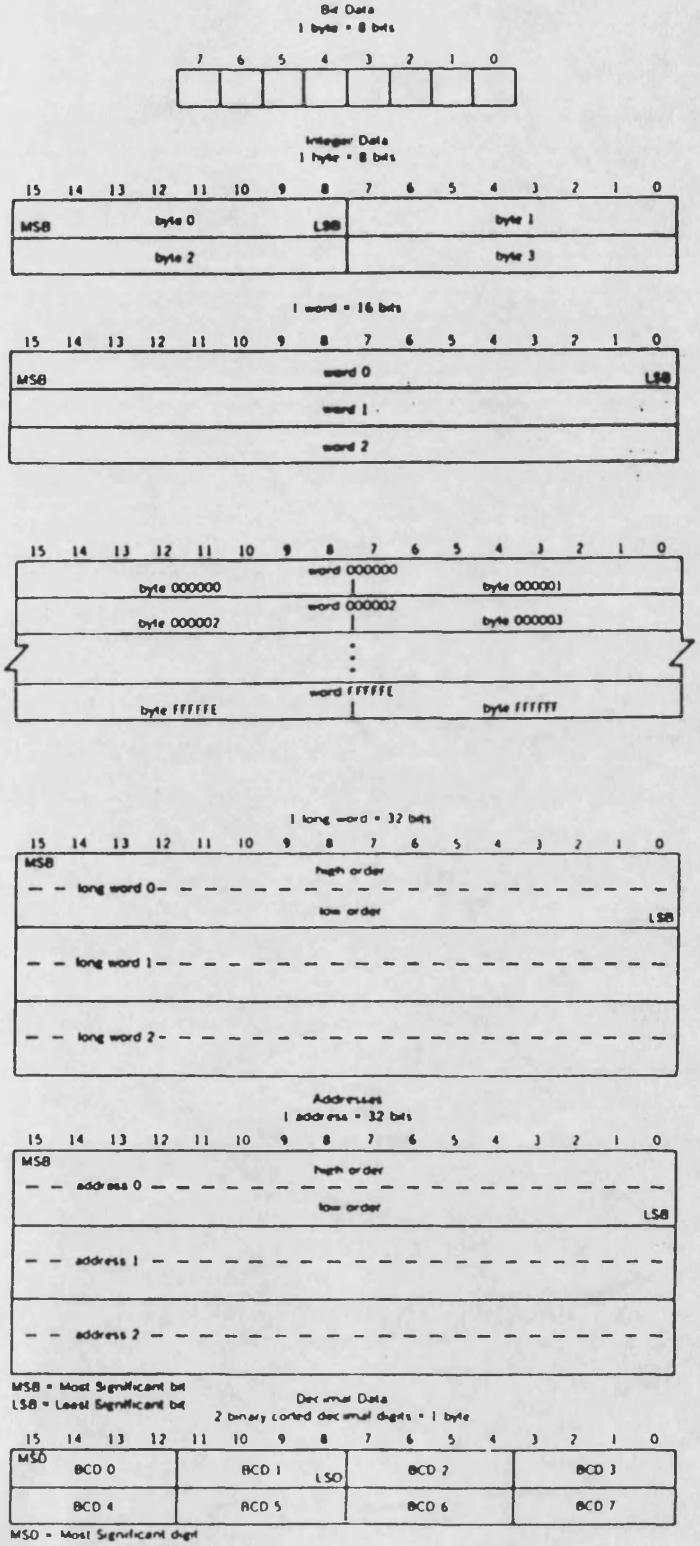


Figure 2.2

The M68000 Exception Vector Table[c2a]

Vector Number(s)	Address Dec	Hex	Space	Assignment
0	0	000	SP	Reset: Initial SSP
	4	004	SP	Reset: Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12-23*	48	030	SD	(Unassigned, reserved)
	95	05F		-
24	96	060	SD	Spurious Interrupt
25-31	96	060	SD	Level 1-7 Interrupt Autovectors
	127	079	SD	-
32-47	128	080	SD	TRAP Instruction Vectors
	191	0BF		-
48-63*	192	0C0	SD	(Unassigned, reserved)
	255	0FF		-
64-255	256	100	SD	User Interrupt Vectors
	1023	3FF		-

SSP - Supervisor Stack Pointer, PC - Programme Counter

SP - Supervisor Programme, SD - Supervisor Data

*Vector numbers 12 through 23 and 48 through 63 are reserved for future enhancements by Motorola.

Figure 3.1

The Darkstar Memory Map

Location	Device	Description
000000 - 03FFFF/5FFFFF†	Main Memory	Section 3.1
800000 - 807FFF	EPROM	Section 3.1
83FF01 - 83FF03*	M6850 ACIA 1	Section 2.10.1
83FF21 - 83FF23*	M6850 ACIA 2	Section 2.10.1
840041 - 84005F*	Marksman WDC	Section 2.5
840060 - 840063	SCSI Interface	Section 3.1
840101 - 84010D*	FD179X FDC	Section 2.7
840380 - 8403BF	M68451 MMU 1	Section 2.3
8403C0 - 8403EF	M68451 MMU 2	Section 2.3
840500 - 8405FF	HD68450 DMAC	Section 2.4

†All addresses are in hexadecimal

*Odd addresses only

Figure 3.2

The SBC Memory Map

Location	Device	Description
000000 - 03FFFF/0FFFFF†	Main Memory	Section 3.2.1
800000 - 801FFF	EPROM	Section 3.2.1
83FF41 - 83FF47*	SY6551 ACIA 1	Section 2.10.2
83FF49 - 83FF4F*	SY6551 ACIA 2	Section 2.10.2
840200 - 84023F	M68451 MMU 1	Section 2.3
840240 - 84027F	M68451 MMU 2	Section 2.3
840281 - 8402BF*	M68230 PI/T	Section 2.11
8402C0 - 8402C9	WD279X FDC	Section 2.7
8402E0	Processor Number Register	Section 3.2.2
840300 - 8403FF	HD68450 DMAC	Section 2.4
8403B1 - 84042F*	Processor Control Registers	Section 3.2.2

†All addresses are in hexadecimal

*Odd addresses only

Figure 3.3

SBC Interrupt Priority Levels

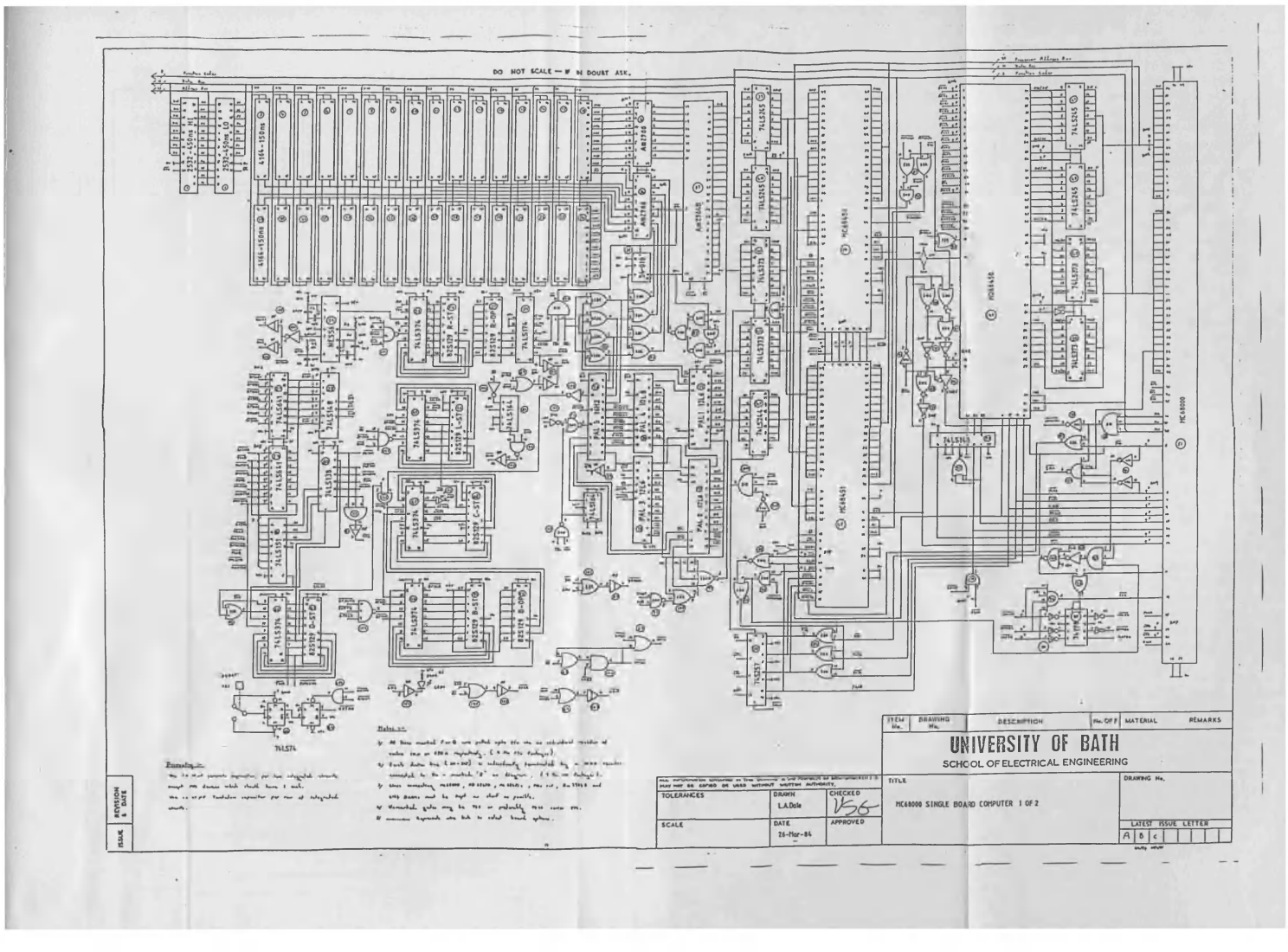
Interrupt Priority Level	Device
7	Abort switch*
6	MMU & PI/T timer
5	Inter-Processor Interrupt*
4	DMAC
3	PI/T ports
2	FDC
1	ACIA's*

*Autovectored

Figure 3.4

Schematic Diagrams of the SBC

The following two pages contain the schematic diagrams of the Single Board Computer.


DO NOT SCALE — IF IN DOUBT ASK.
UNIVERSITY OF BATH
SCHOOL OF ELECTRICAL ENGINEERING
MC68000 SINGLE BOARD COMPUTER 1 OF 2
L.A.Dale
26-Mar-84
ISSUE
REVISION
DRAWING No.
LATEST ISSUE LETTER
TOLERANCES
SCALE
DRAWN
CHECKED
APPROVED
DATE
TITLE
DESCRIPTION
No. OFF
MATERIAL
REMARKS

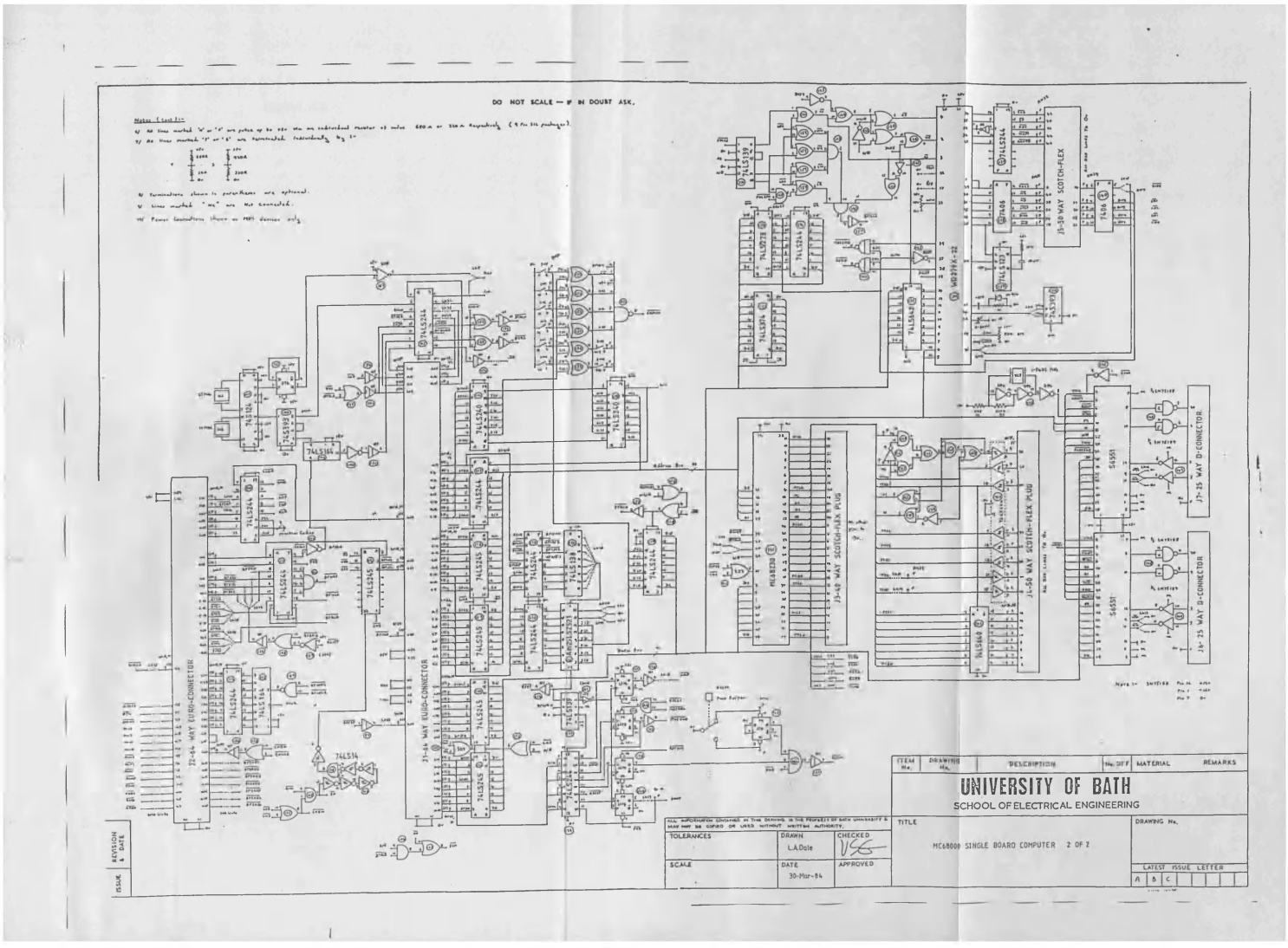

DO NOT SCALE — IF IN DOUBT ASK.
Notes (cont):-
J2-64 WAY EURO-CONNECTOR
J1-64 WAY EURO-CONNECTOR
J3-48 WAY SCOTCH-HLX PLUG
J4-50 WAY SCOTCH-FLEX PLUG
J5-30 WAY SCOTCH-FLEX
J7-25 WAY D-CONNECTOR
J6-25 WAY D-CONNECTOR
ITEM No.
DRAWING No.
DESCRIPTION
No. OFF
MATERIAL
REMARKS
UNIVERSITY OF BATH
SCHOOL OF ELECTRICAL ENGINEERING
TOLERANCES
DRAWN
L.A.Dale
CHECKED
SCALE
DATE
30-Mar-84
APPROVED
TITLE
MC68000 SINGLE BOARD COMPUTER 2 OF 2
DRAWING No.
LATEST ISSUE LETTER
A B C
ISSUE
REVISION & DATE

Figure 3.5

The Single Board Computer

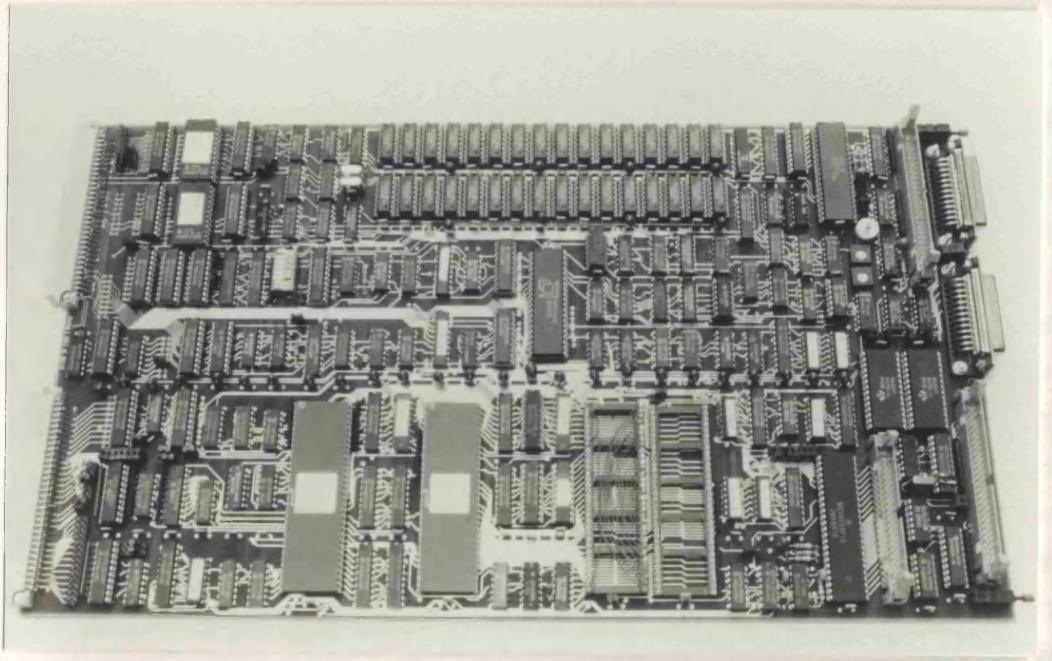


Figure 5.1

Operator Precedence in C[22]

Listed below are the operators in the C programming language in order of precedence. Precedence decreases down the table and operators listed on the same line have equal precedence.

Operator	Associativity
() [] -> .	left to right
! ~ ++ - (*type*) * & sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= etc.	right to left
,	left to right

Figure 8.1

Comparative Results Using Eratosthenes' Sieve

Running Processes	Efficiency Ratio*	
	SPOS	MPOS
1 Process	0.51	1.00
4 Processes	0.13	0.55
3 Processes + 1 Suproc†	N/A	1.00

SPOS = Single Processor Operating System (i.e. standard UNIX V/68)

MPOS = Multi-Processor Operating System

*Efficiency Ratio = (Processor Usage) ÷ (Total Elapsed Time)

†Timing done for the suproc only